



University of
Pittsburgh

Algorithms and Data Structures 2

CS 1501



Spring 2023

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
 - Homework 6: this Friday @ 11:59 pm
 - Lab 4: Tuesday 2/21 @ 11:59 pm
 - Assignment 2: Friday 3/17 @ 11:59 pm
 - Support video and slides on Canvas
- Lost points because autograder or simple mistake?
 - please reach out to Grader TA over Piazza
- Navigating the Panopto Videos
 - Video contents
 - Search in captions

Previous lecture

- R-way Radix Search Trie
- De La Braindais (DLB) Trie

This Lecture

- More on DLB Trie
- Huffman Compression

Solution: De La Briandais tries (DLBs)

Main idea: replace the array inside the node of the R-way trie with a linked-list

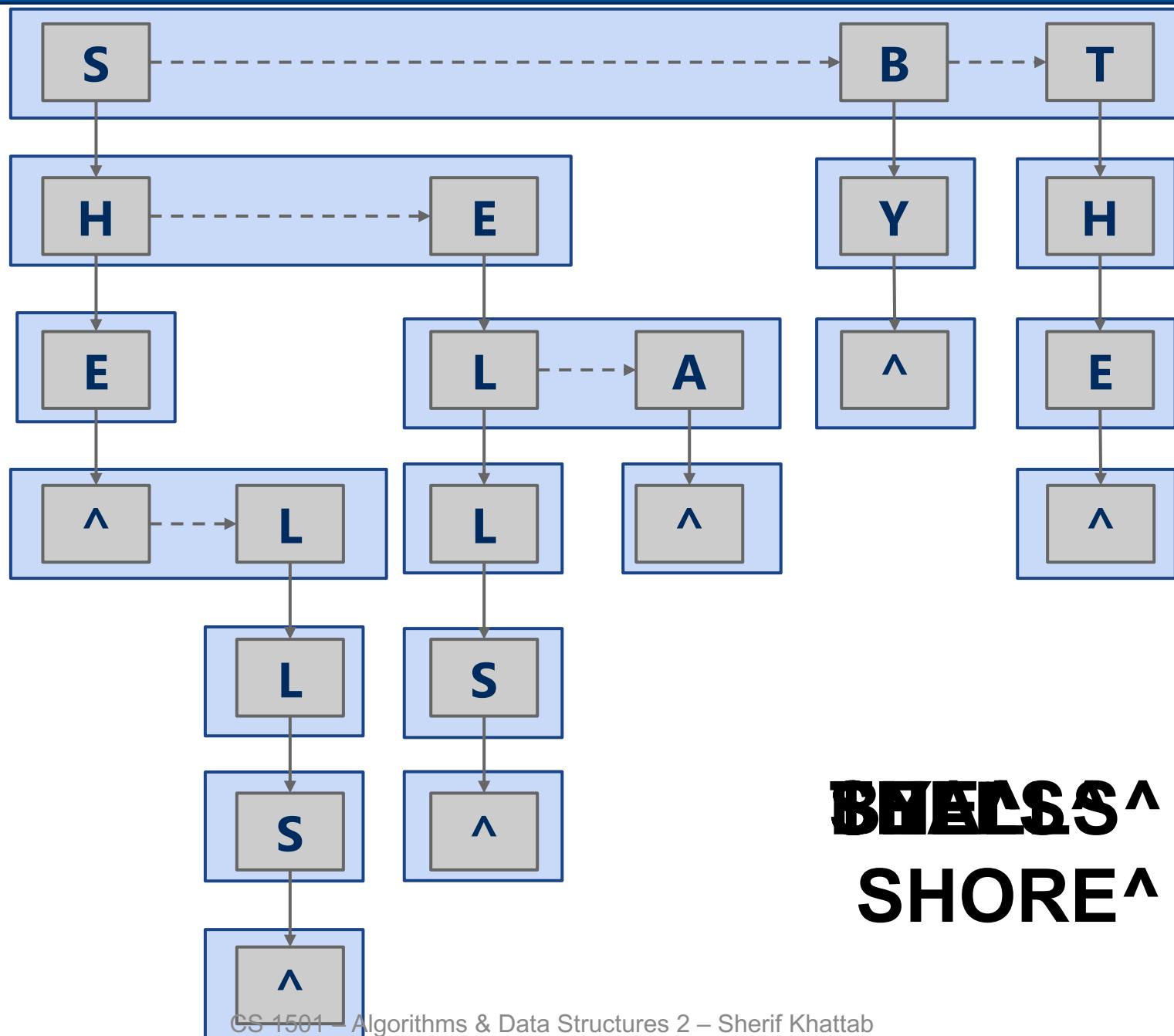
De La Briandais (DLB) Trie

- Tree-like structure used for searching when keys are sequences of characters
- Each ***nodelet***
 - stores one character
 - points to a sibling (forming a linked list of siblings)
 - points to a child

Adding to DLB Trie

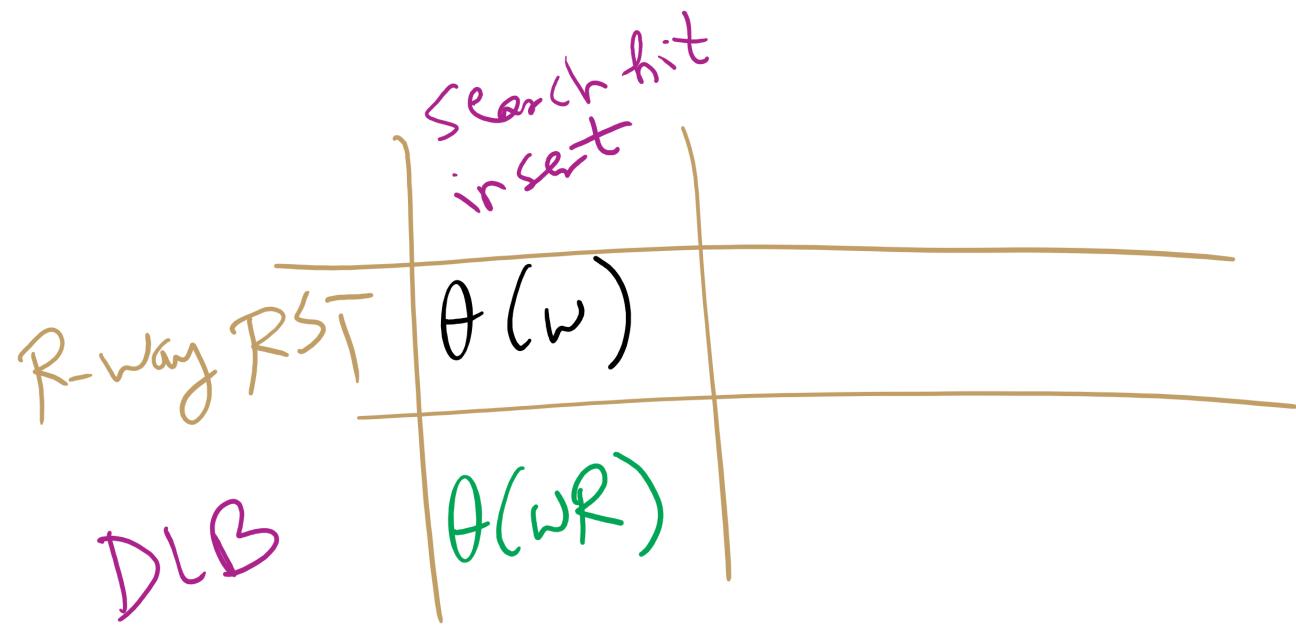
- if root is null, set $\text{root} \leftarrow \text{new node}$
- current node $\leftarrow \text{root}$
- for each *character c* in the key
 - Search for *c* in linked list headed at current (sibling links)
 - if not found, new node, put *c*, and attach to linked list of siblings
 - move to child of the found/new node
 - either recursively or by $\text{current} \leftarrow \text{node.child}$
- if at last character of key, insert value and return

DLB Example



DLB analysis

- How does DLB performance differ from R-way tries?
- Which should you use?



De La Briandais (DLB) Trie

- worst-case running time is $O(wR)$
 - w: number of characters in the key
 - R: alphabet size
- worst-case can be avoided by using DLB only when the sibling lists are short

DLB vs. R-way RST: Space comparison example

- **Q: How does DLB save space over R-way RST?**

Assume the following set of keys:

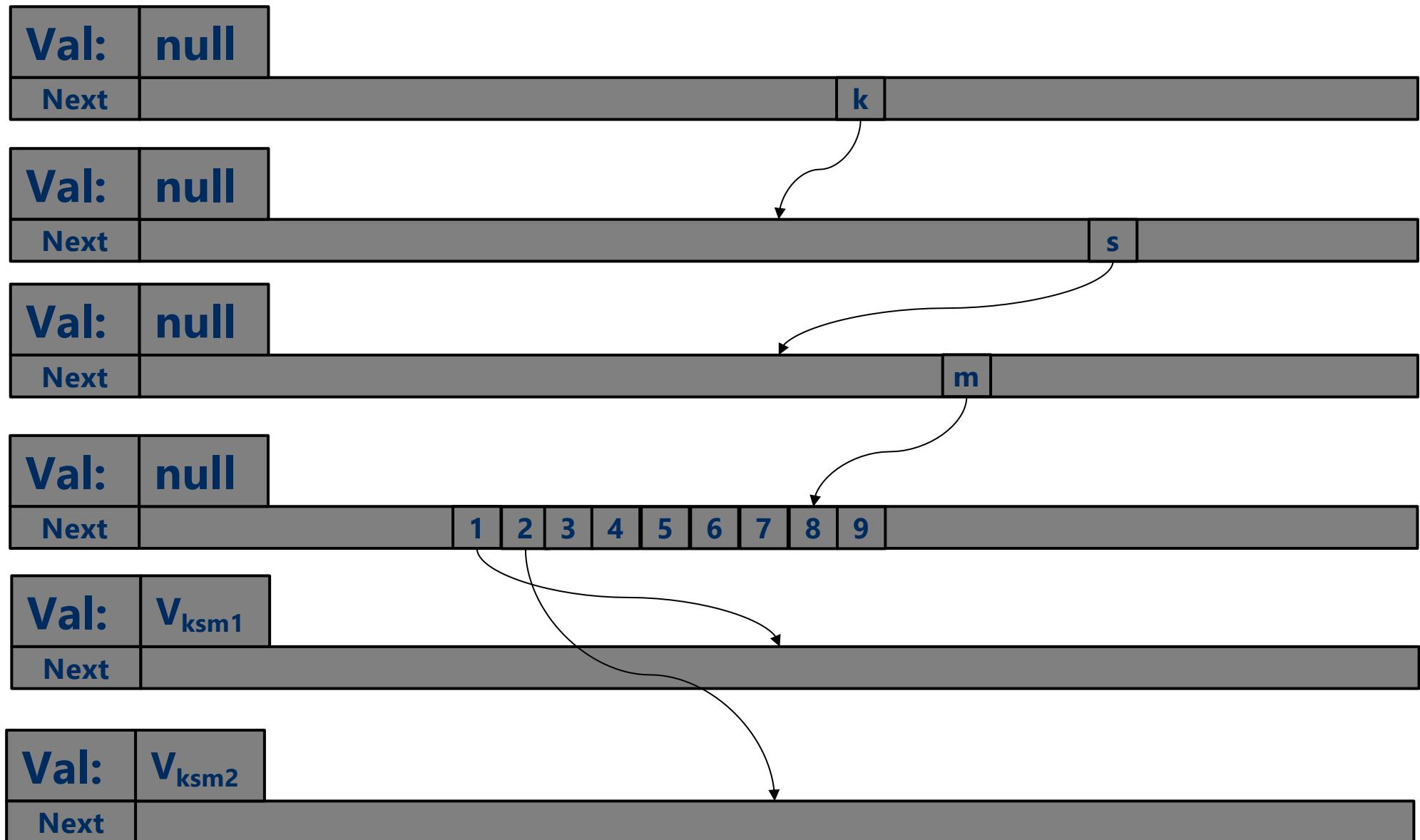
- ksm1 ... ksm9
- How big does an 256-way RST take vs. a DLB trie?

R-way RST

```
private class Node {  
    private Object val;  
    private Node[] next;  
  
    private Node(){  
        next = new Node[R];  
    }  
}
```

Each node takes $4*(R+1) = 4*257 = 1028$ bytes,
assuming 4 bytes per reference variable

R-way RST



R-way RST

We will end up with $4 + 9 = 13$ nodes

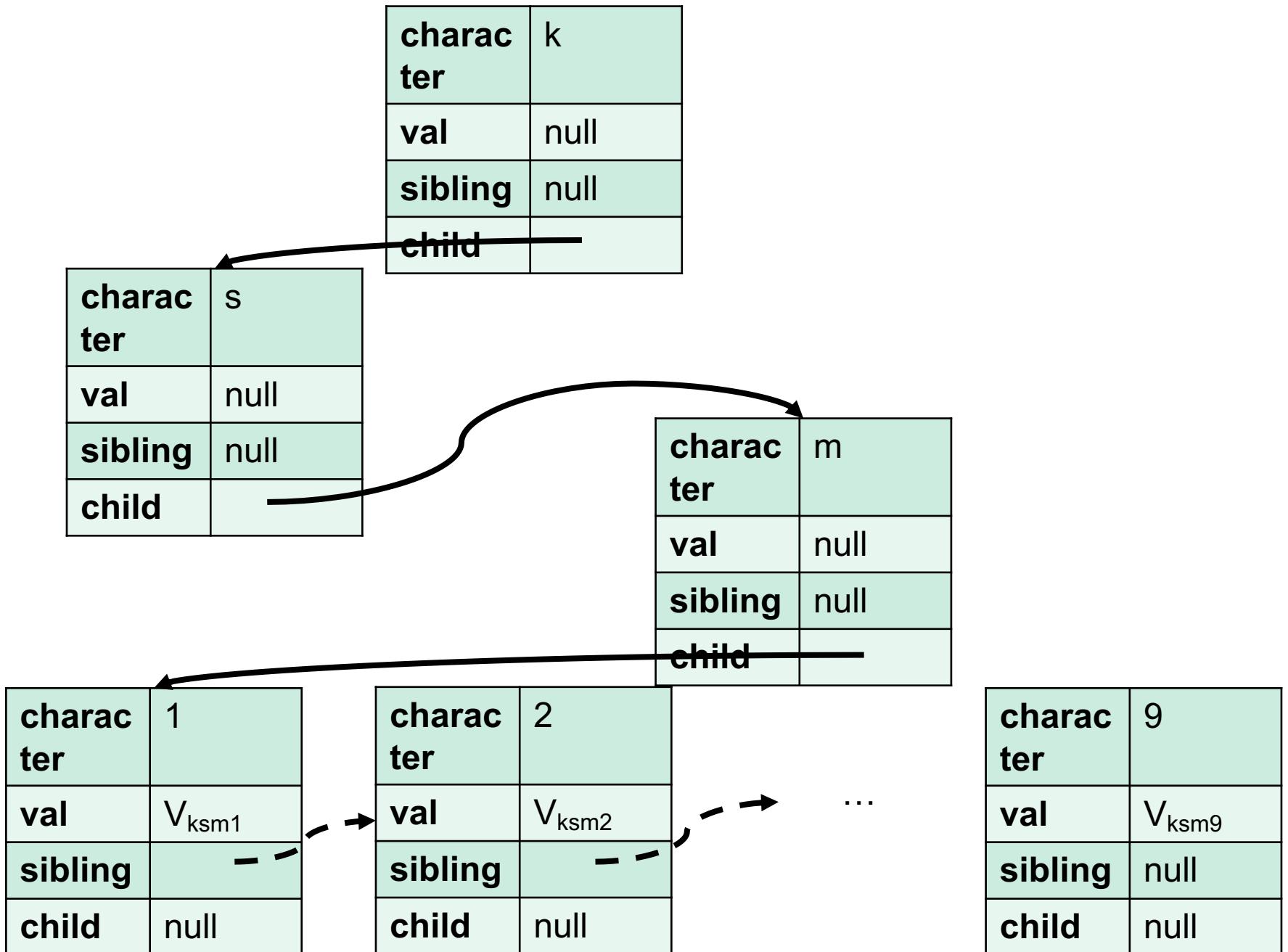
Total space is $13 * 1028 = 13,364$ bytes

DLB Trie

```
private class DLBNode<T> {  
    private Character character;  
    private Object val;  
    private Node sibling;  
    private Node child;  
}
```

Each node takes $4*4 = 16$ bytes, assuming 4 bytes per reference variable

DLB Trie



DLB Trie

We will end up with 12 nodes

Total space is $12 * 16 = 192$ bytes

Compare to 13,364 bytes with an R-way RST

Runtime Comparison for Search Trees/Tries

	Search hit	Search miss <i>(average)</i>	insert
BST	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
RB-BST	$\Theta(\log n)$	$\Theta(\log \alpha)$	$\Theta(\log n)$
DST	$\Theta(b)$	$\Theta(\log n)$	$\Theta(b)$
RST	$\Theta(b)$	$\Theta(\log n)$	$\Theta(b)$
R-way RST	$\Theta(w)$	$\Theta(\log_R n)$	$\Theta(w)$
DLB	$\Theta(wR)$	$\Theta(\log_R n \cdot R)$	$\Theta(w \cdot R)$

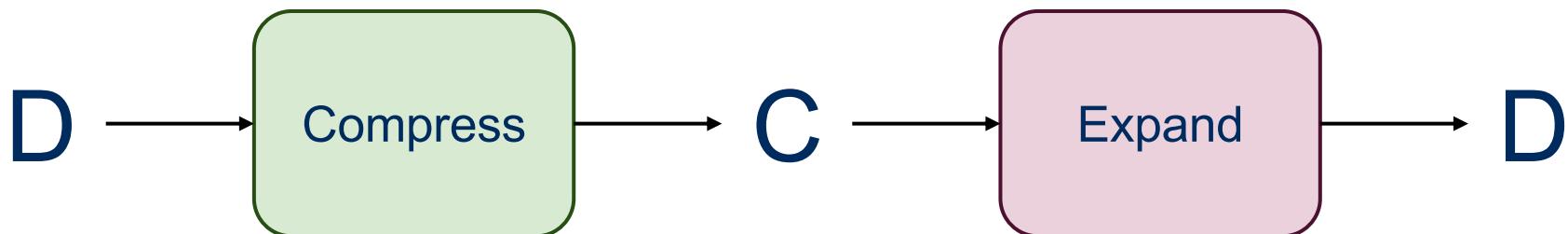
Final notes on Search Tree/Tries

- We did not present an exhaustive look at search trees/tries, just the sampling that we're going to focus on
- Many variations on these techniques exist and perform quite well in different circumstances
 - Ternary search Tries
 - R-way tries without 1-way branching
- See the table at the end of Section 5.2 of the text

Problem of the Day: Compression

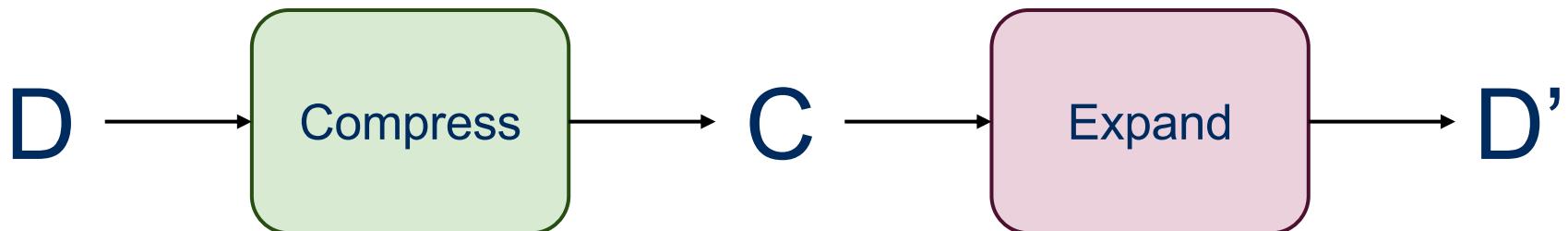
- Input: A file containing a sequence of n characters
 - each character encoded as an 8-bit Extended ASCII
 - total file size = $8*n$ bits
- Output: A shorter bitstring
 - of bitlength $< 8*n$
 - such that the original sequence can be fully restored from the shorter bitstring

Lossless Compression



- Input can be recovered from compressed data exactly
- Examples:
 - zip files, FLAC

Lossy Compression



- Information is permanently lost in the compression process
- Examples:
 - MP3, H264, JPEG
- With audio/video files this typically isn't a huge problem as human users might not be able to perceive the difference

Lossy examples

- MP3
 - “Cuts out” portions of audio that are considered beyond what most people are capable of hearing
- JPEG



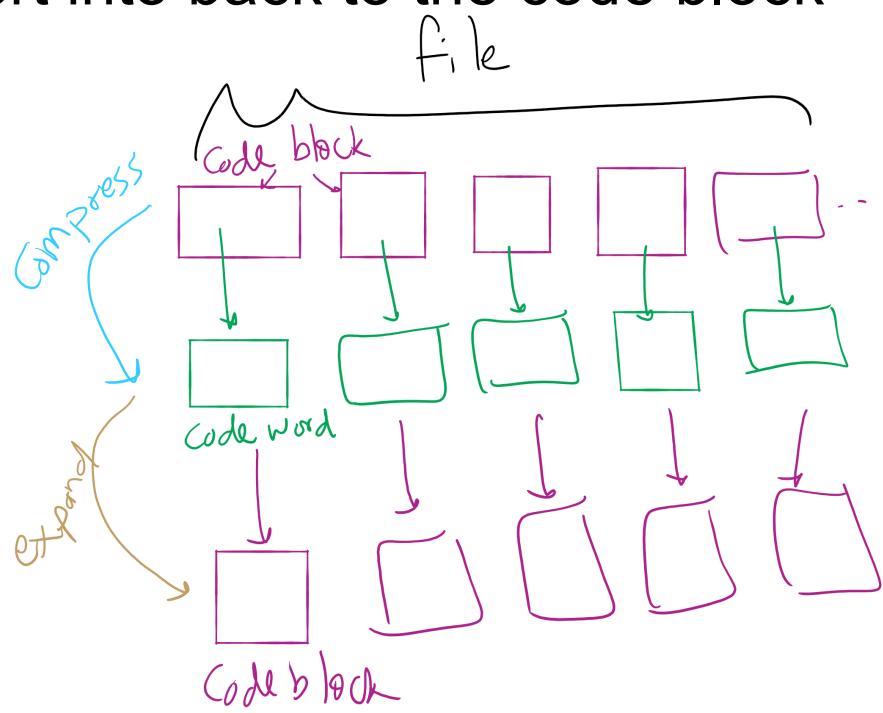
40K



28K

Lossless Compression Framework

- Compression:
 - For each code block in the input file
 - Convert into a codeword
- Expansion:
 - For each codeword in the compressed file
 - Convert into back to the code block



Cases of the Lossless Compression Framework

- **Case 1: fixed-size code blocks, fixed-size codewords**
- To achieve compression, codeword size has to be less than code block size
 - Can be done if the used alphabet in the file is smaller than the Extended ASCII
 - e.g., a file of English letters
 - can use 5 bits per codeword instead of 8 bits
 - **Compression ratio** = 5/8
 - e.g., a file of 64-encoded characters
 - lower-case and upper-case English letters, digits, and two symbols
 - can use 6 bits per codeword
 - Typically, the reduced alphabet is stored at the beginning of the compressed file
 - increase the compressed file size and the compression ratio

Cases of the Lossless Compression Framework

- **Case 2: fixed-size code blocks, variable-size codewords**
- To achieve compression, assign shorter codewords to more frequently occurring characters
- Total compressed file size
 - R : alphabet size
 - f_i : frequency of character i
 - $|\text{codeword}_i|$: bitlength of codeword corresponding to character i
 - compressed file size = $\sum_{i=1}^R f_i |\text{codeword}_i|$
- *Huffman compression* is an example of this case

Cases of the Lossless Compression Framework

- **Case 3: variable-size code blocks, fixed-size codewords**
- To achieve compression, find recurring patterns that are *as long as possible* in the input file and assign fixed-size codewords to each
- *LZW compression* and *run-length encoding* are examples of this case

Cases of the Lossless Compression Framework

- **Case 4: variable-size code blocks, variable-size codewords**
- Typically implemented as Case 3 followed by Case 2

Solution 1: Huffman Compression

- What if we used *variable length* codewords instead of the constant 8 bits per character? Could we store the same info in less space?
 - Different characters are represented using codes of different bit lengths
 - What about different usage frequencies between characters?
 - In English, R, S, T, L, N, E are used much more than Q or X
 - Using this, we can achieve compression by:
 - Using fewer bits to represent more common characters
 - Using longer codes to represent less common characters

But we have to worry about restoring the data!

- Decoding was easy for block codes
 - Grab the next 8 bits in the bitstring
 - How can we decode a bitstring that is made of variable length code words?
 - use delimiters?
 - BAD example of variable length encoding:

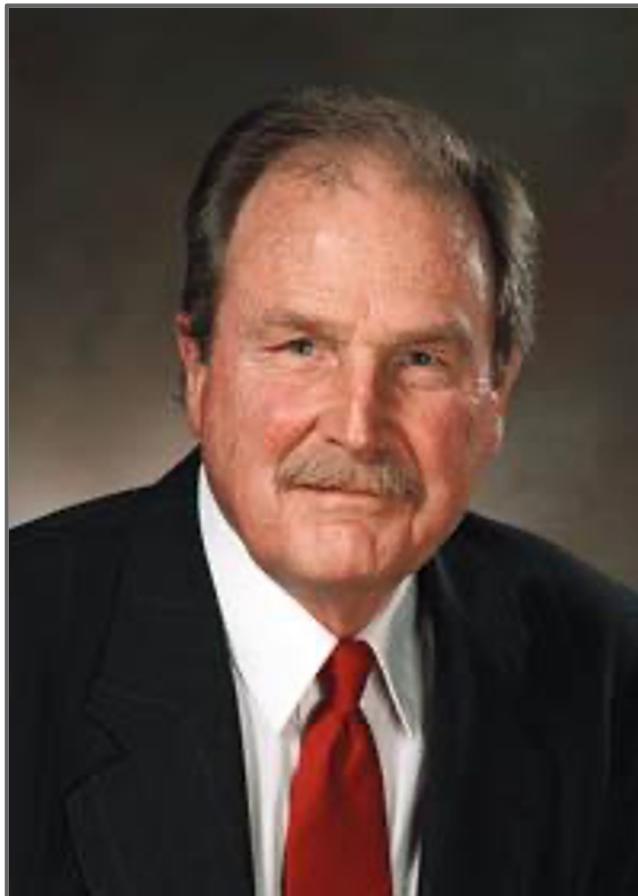
00	T
01	K
001	U
100	R
101	C
10101	N

Variable length encoding for lossless compression

- Codes must be *prefix free*
 - No code can be a prefix of any other in the scheme

How can we create these prefix-free codes?

Huffman encoding!



Subproblem: Prefix-free Compression

- Input: A sequence of n characters
- Output: A codeword h_i for each character i such that
 - No codeword is a prefix of any other
 - When each character in the input sequence is replaced with each codeword
 - the length of the compressed sequence is minimum (over other fixed-size code-block encoding schemes)
 - the original sequence can be fully restored from the compressed bitstring

Generating Huffman codes

- Assume we have K characters that are used in the file to be compressed and each has a weight (its frequency of use)
- Create a forest, F , of K single-node trees, one for each character, with the single node storing that char's weight
- while $|F| > 1$:
 - Select $T_1, T_2 \in F$ that have the smallest weights in F
 - Create a new tree node N whose weight is the sum of T_1 and T_2 's weights
 - Add T_1 and T_2 as children (subtrees) of N
 - Remove T_1 and T_2 from F
 - Add the new tree rooted by N to F

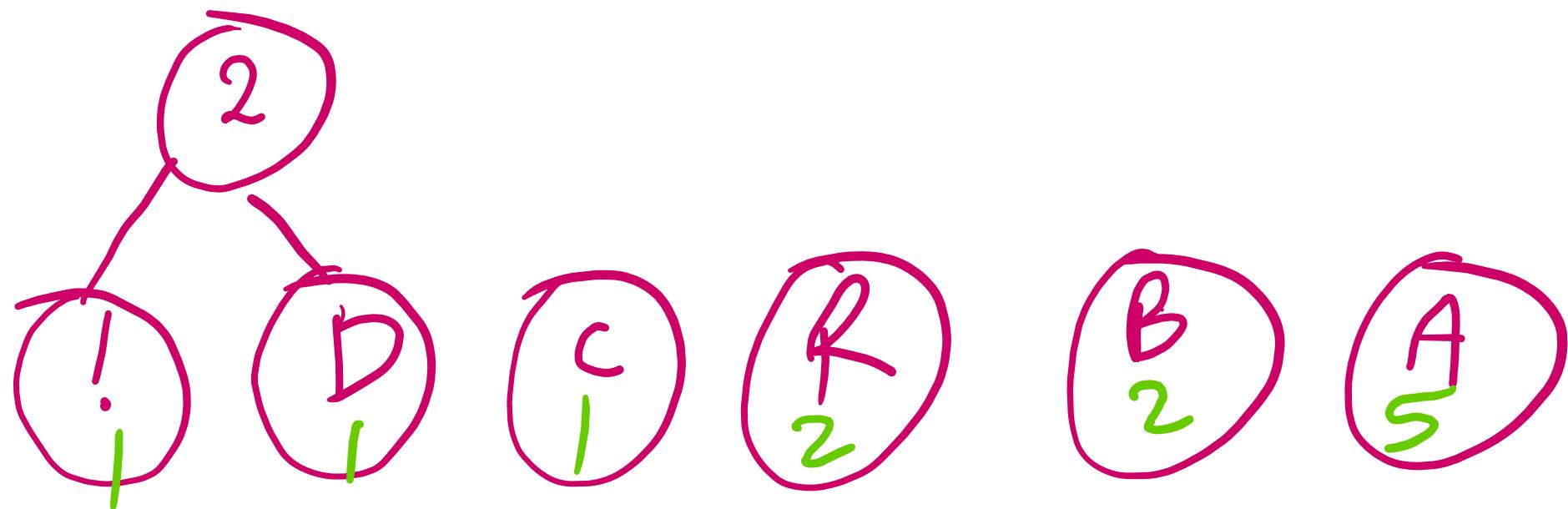
Example

- Build a tree for “ABRACADABRA!”
- file size: $n = 12$
- no. of unique characters: $K = 6$



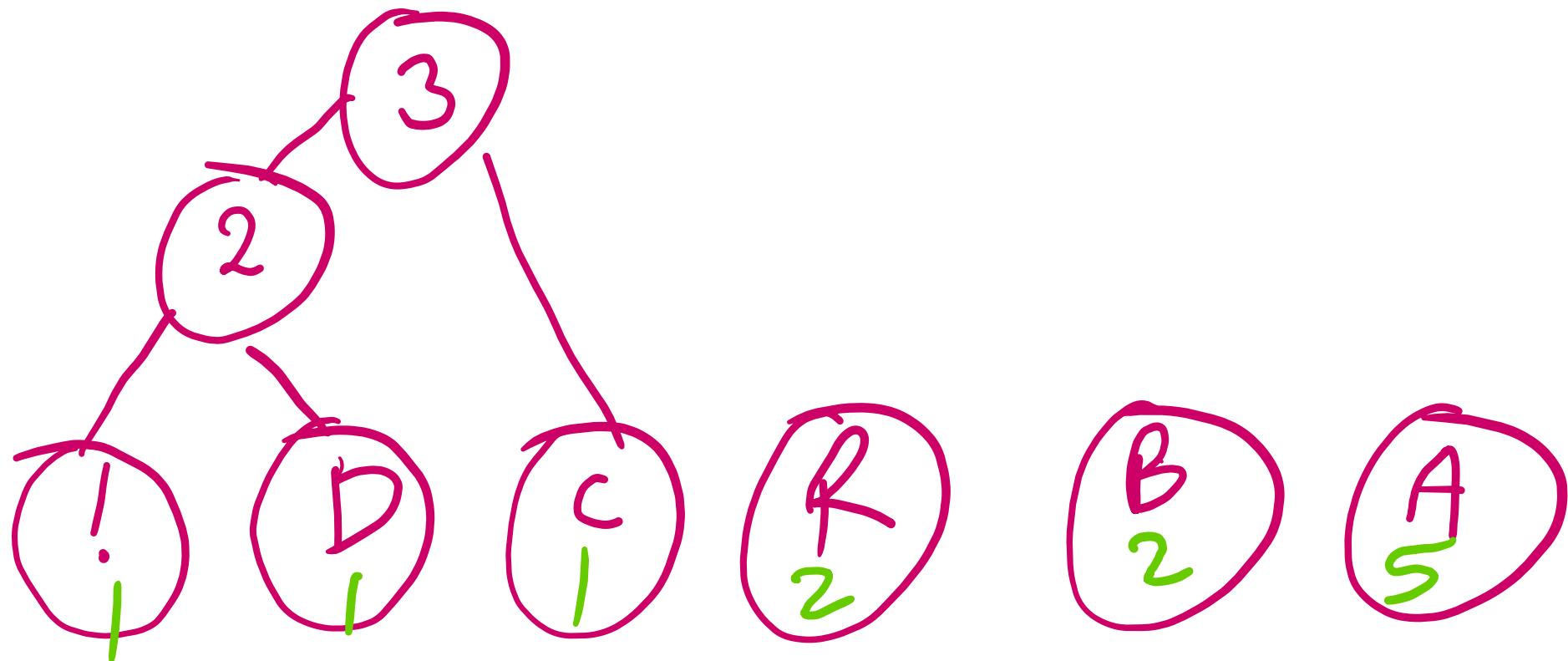
Example

- Build a tree for “ABRACADABRA!”



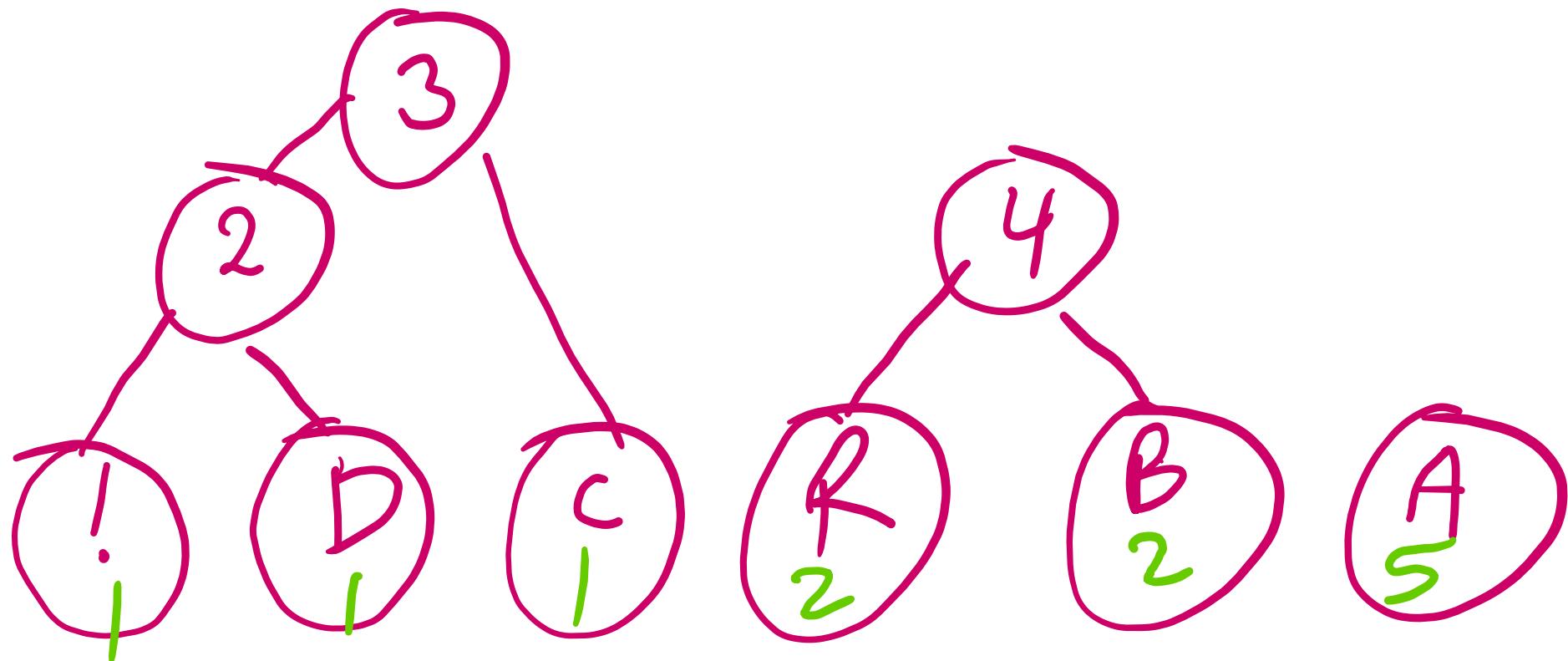
Example

- Build a tree for “ABRACADABRA!”



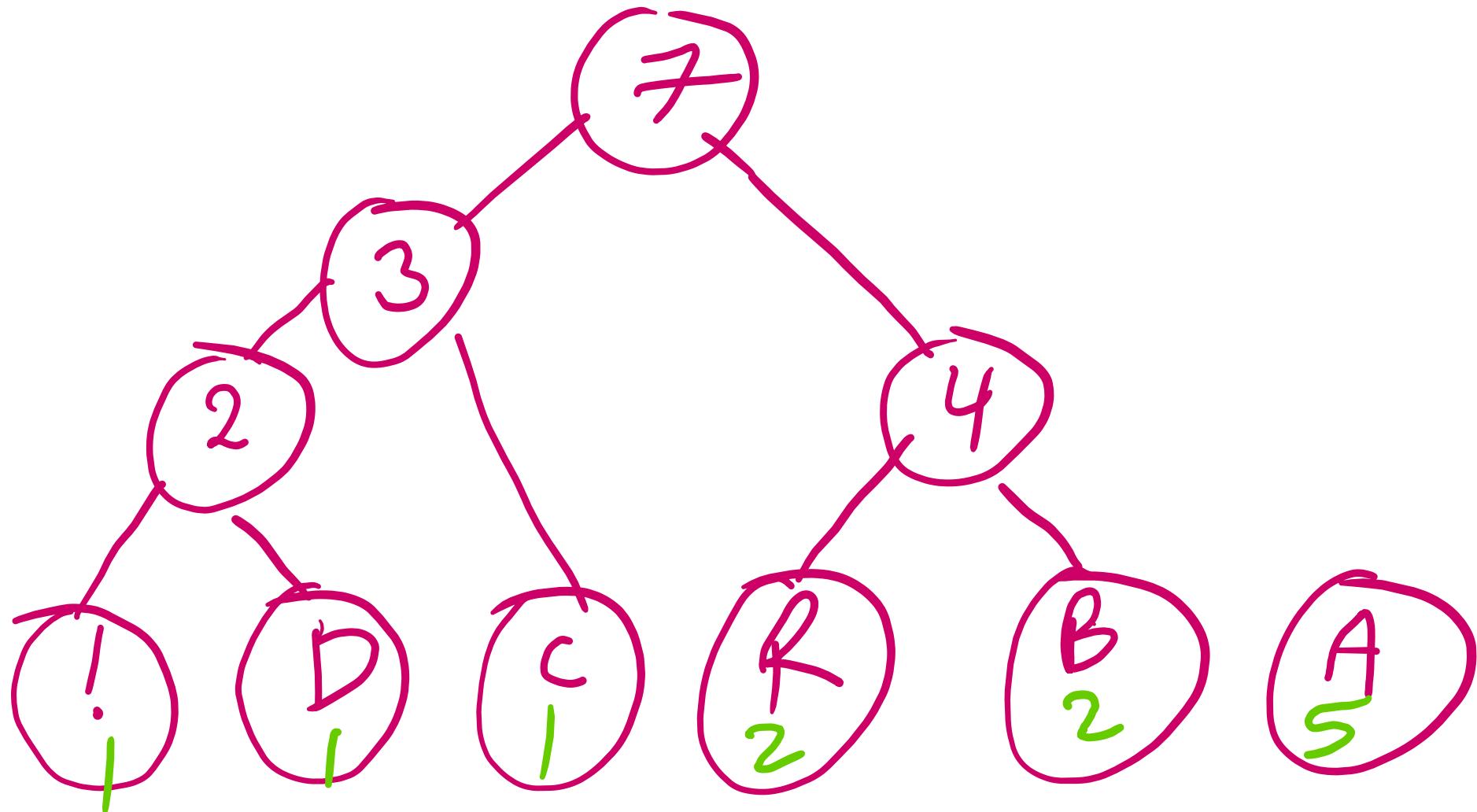
Example

- Build a tree for “ABRACADABRA!”



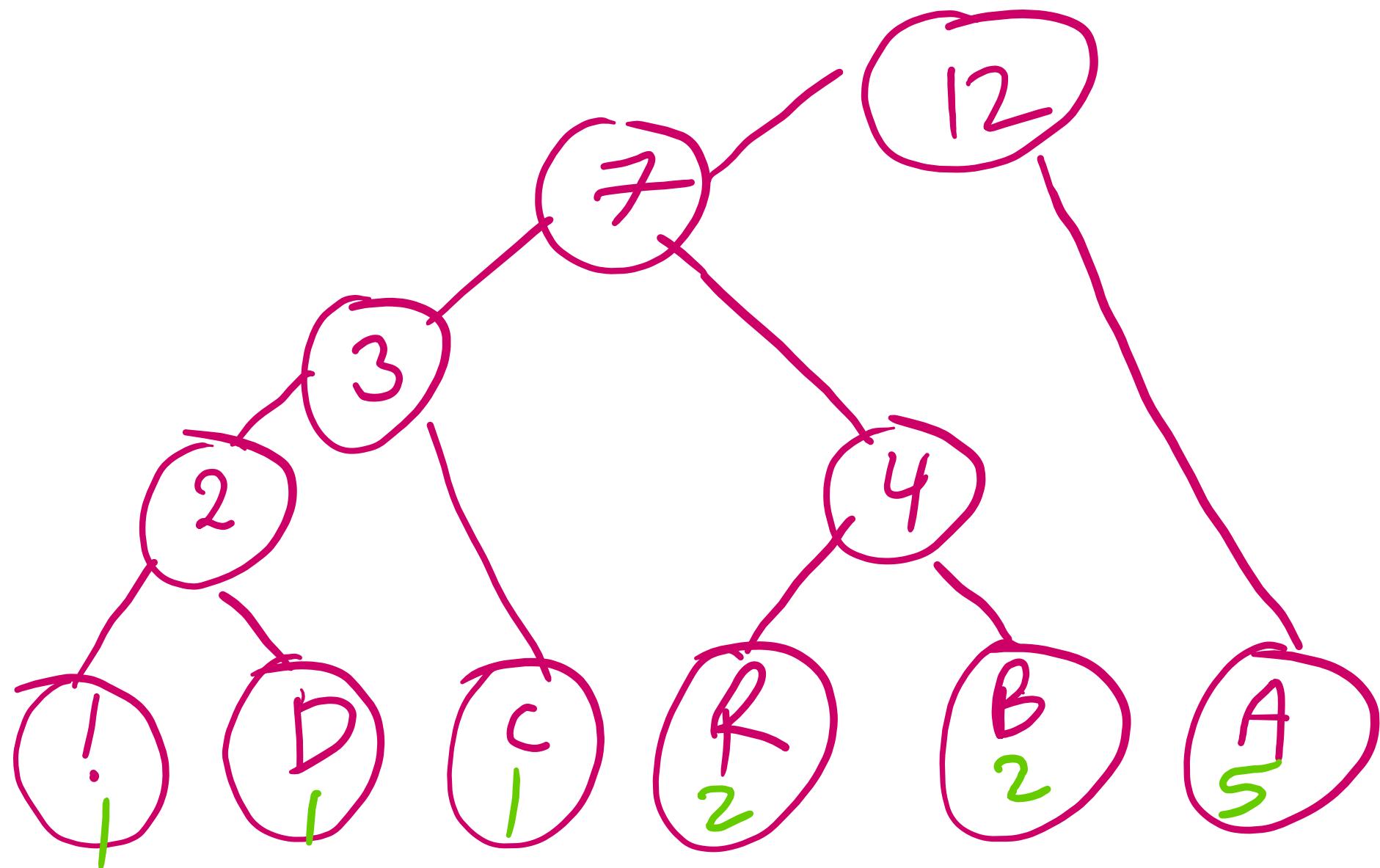
Example

- Build a tree for “ABRACADABRA!”



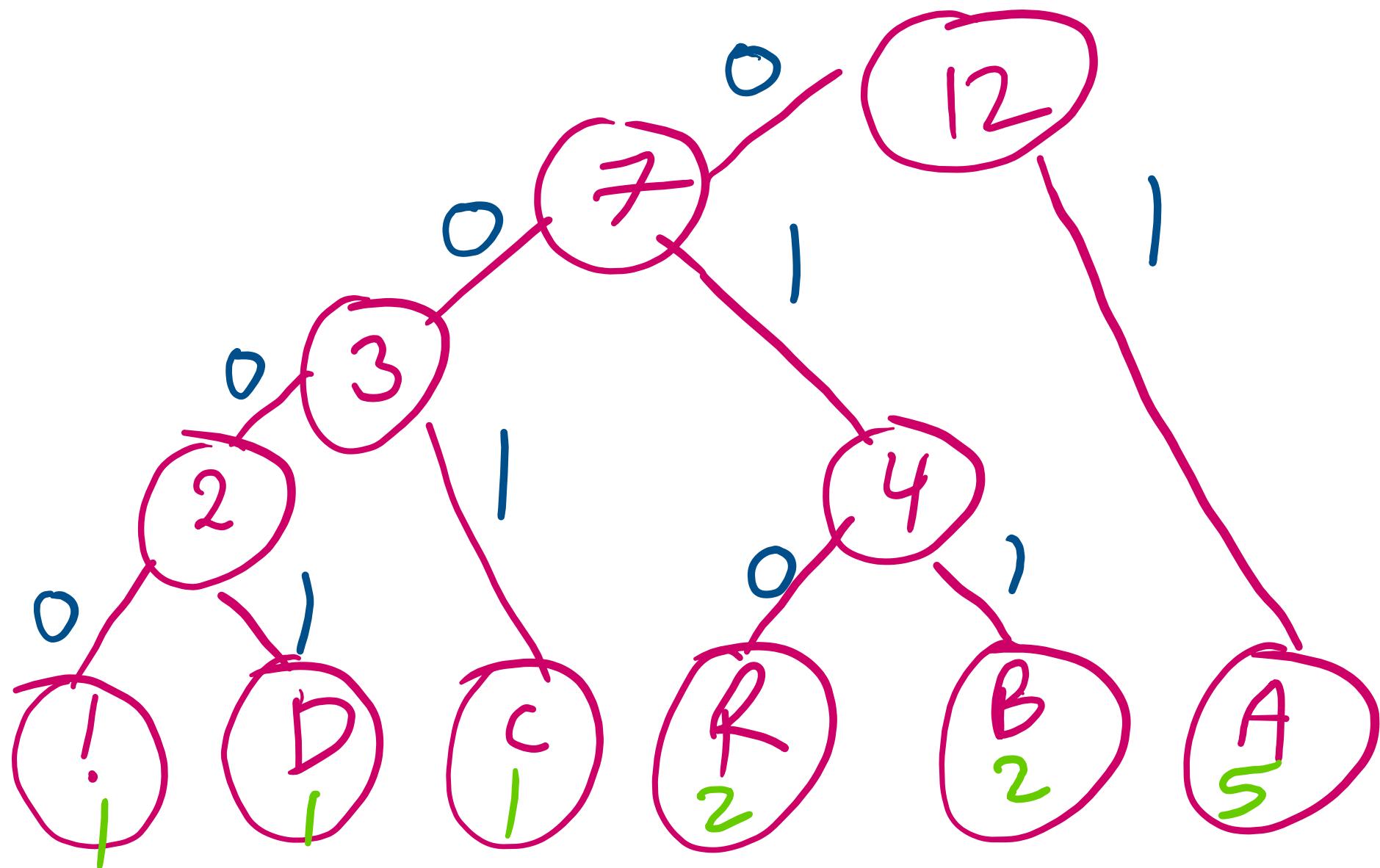
Example

- Build a tree for “ABRACADABRA!”



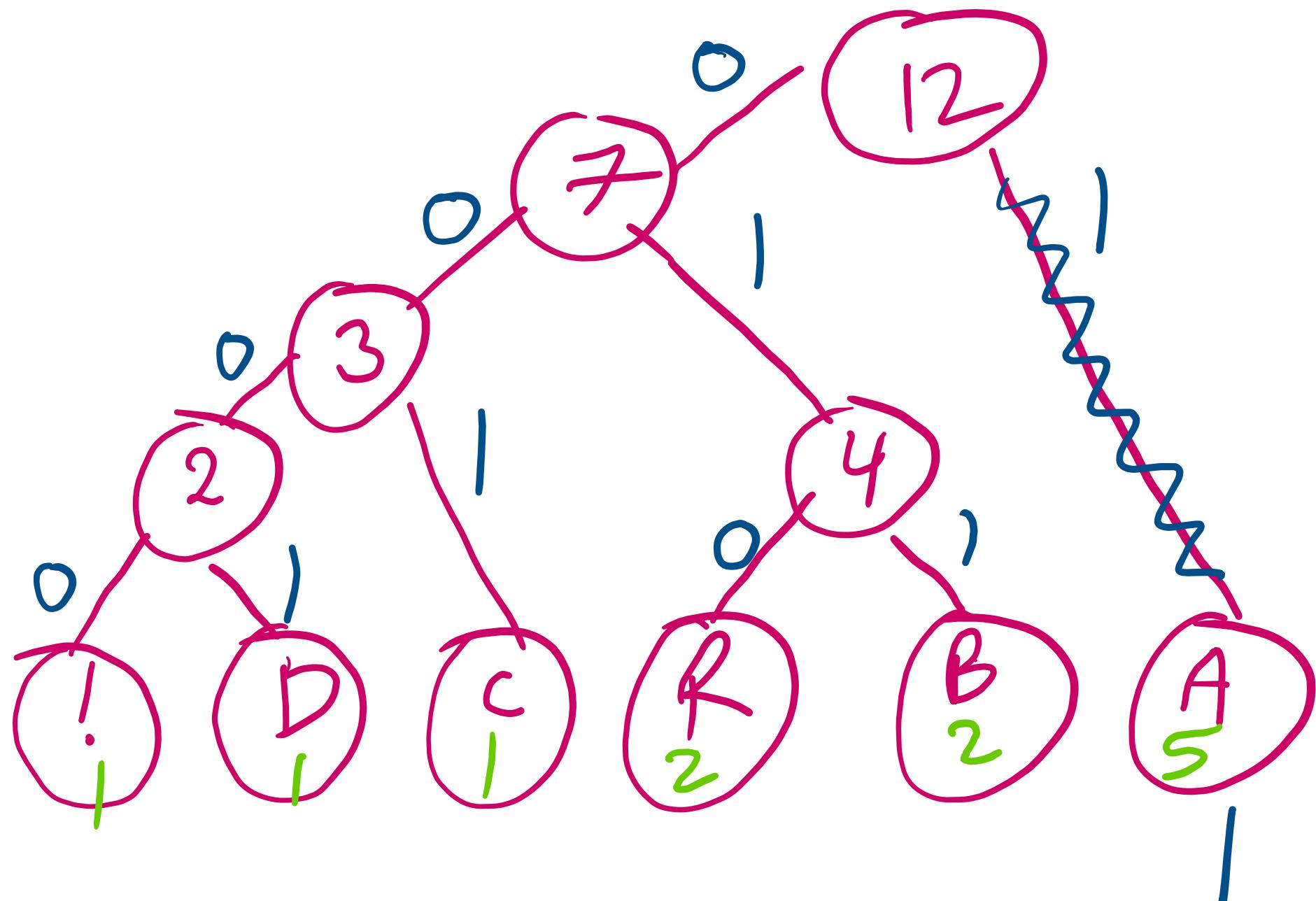
Example

- Build a tree for “ABRACADABRA!”



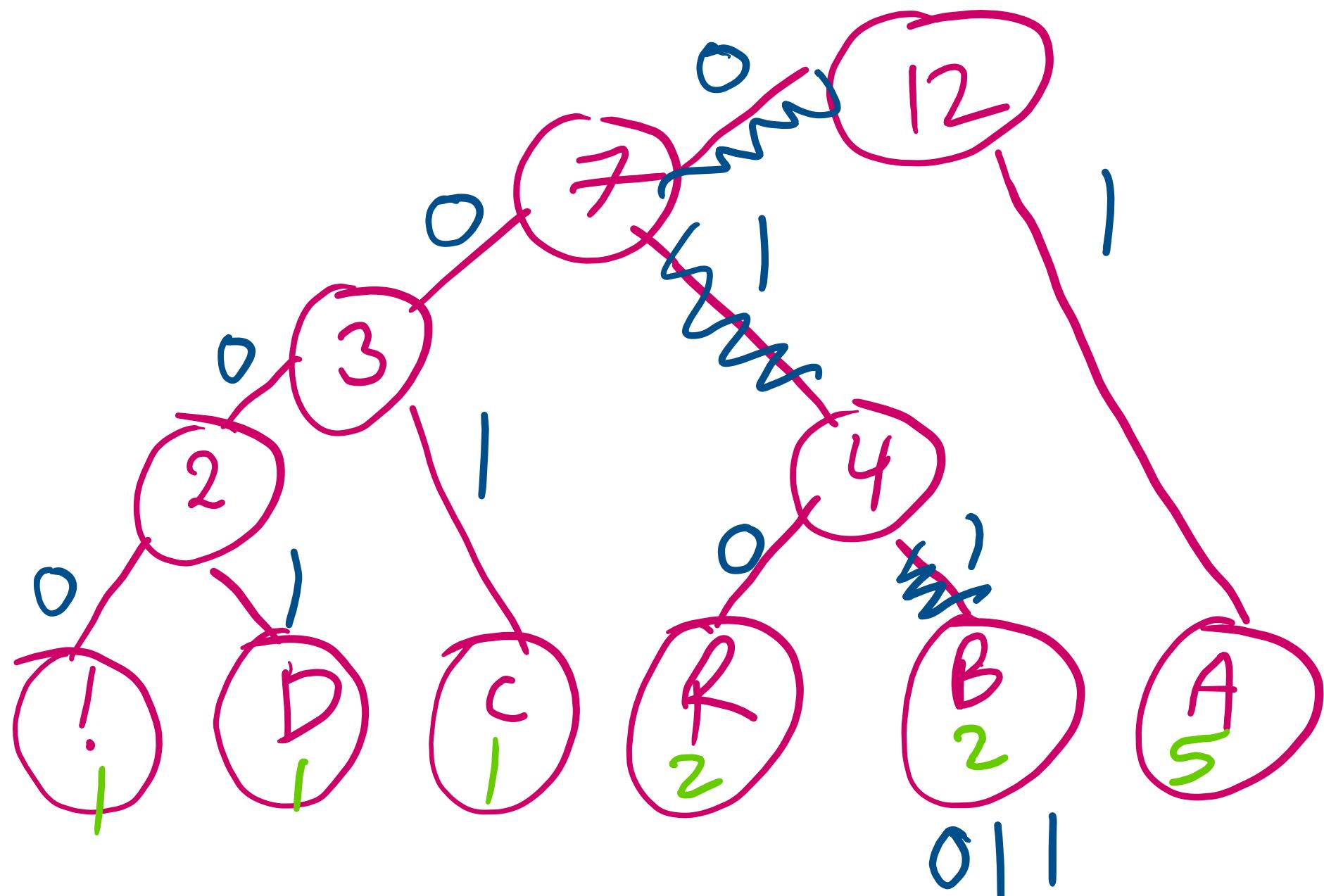
Example

- Build a tree for “ABRACADABRA!”



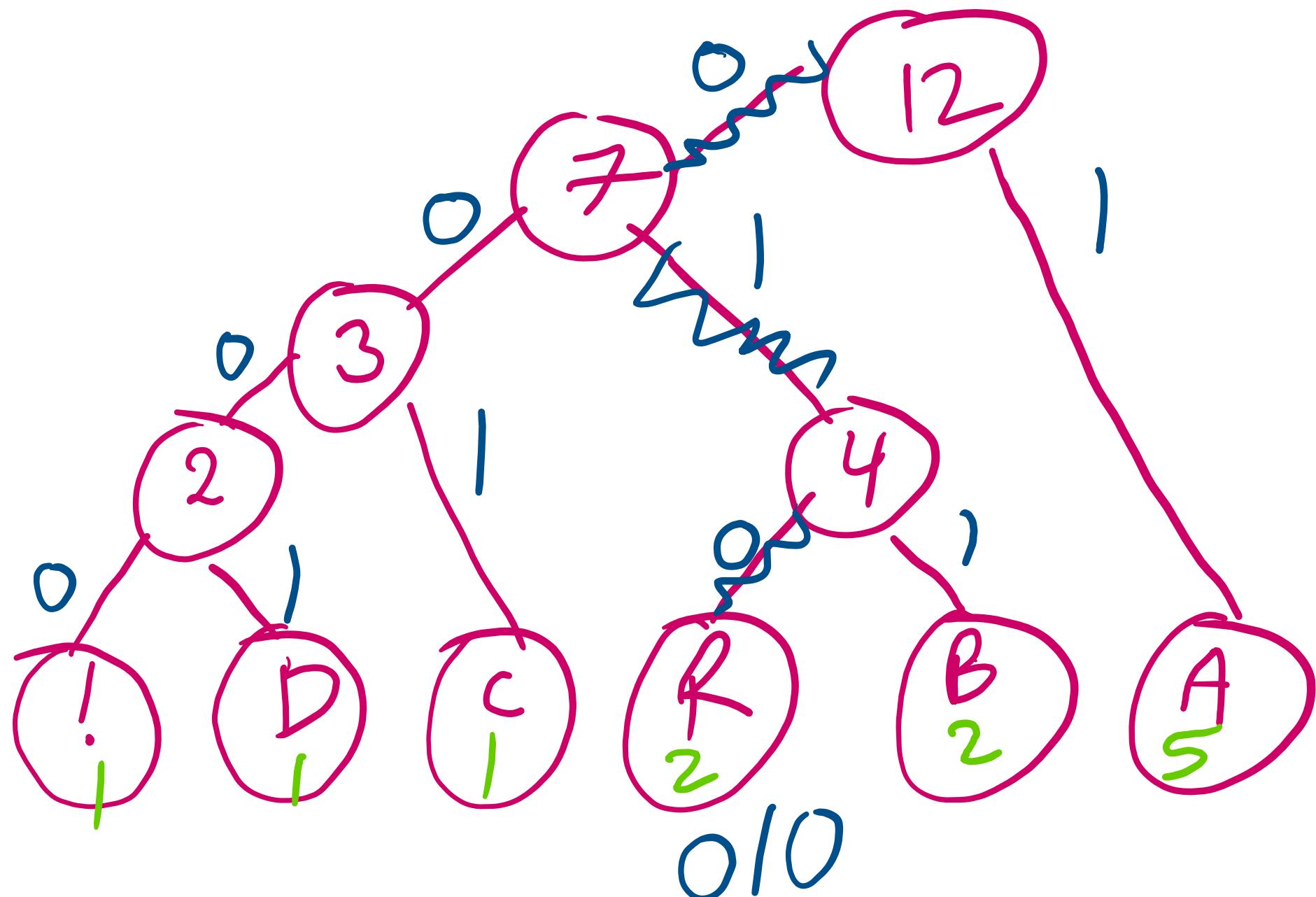
Example

- Build a tree for “ABRACADABRA!”



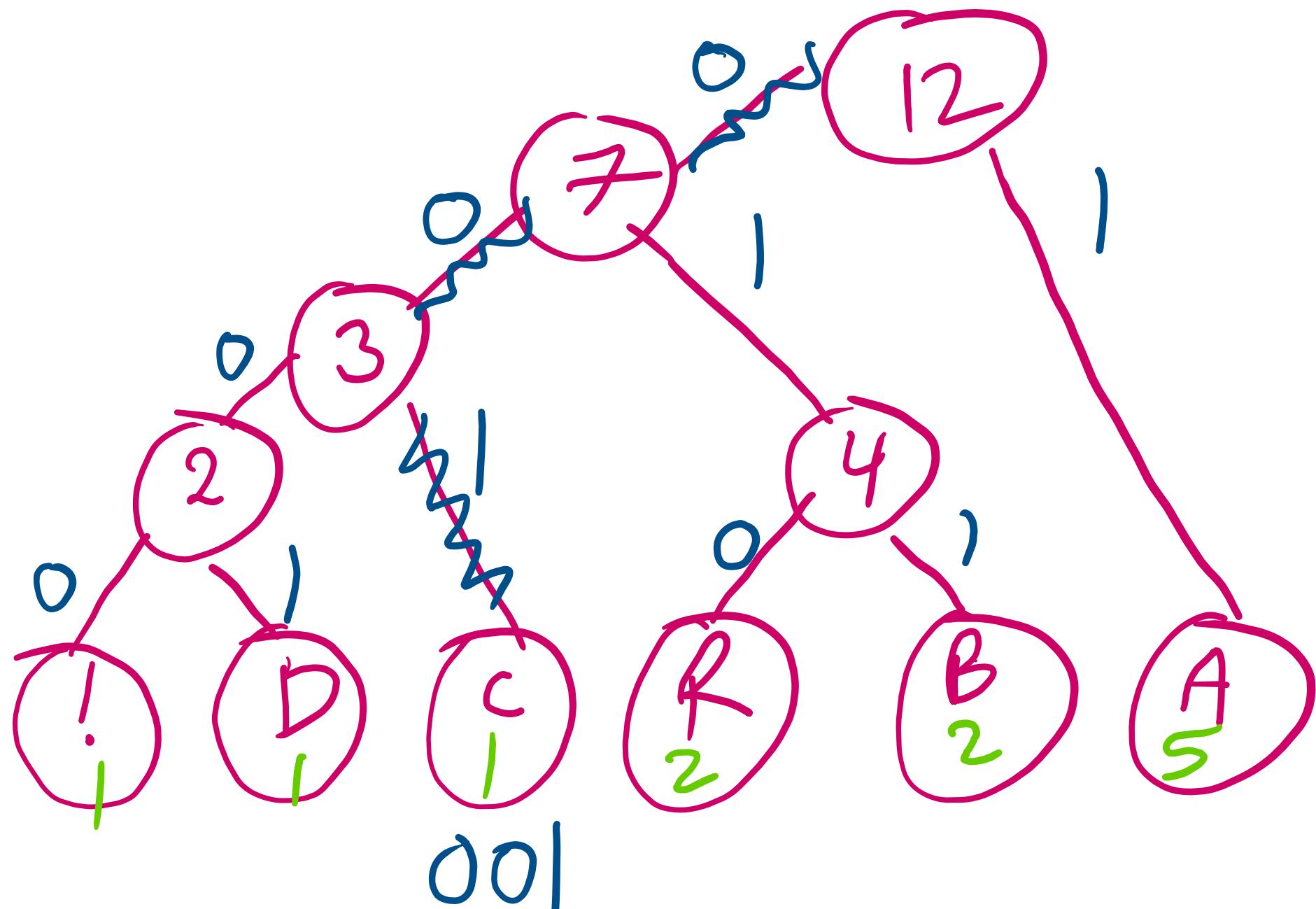
Example

- Build a tree for “ABRACADABRA!”



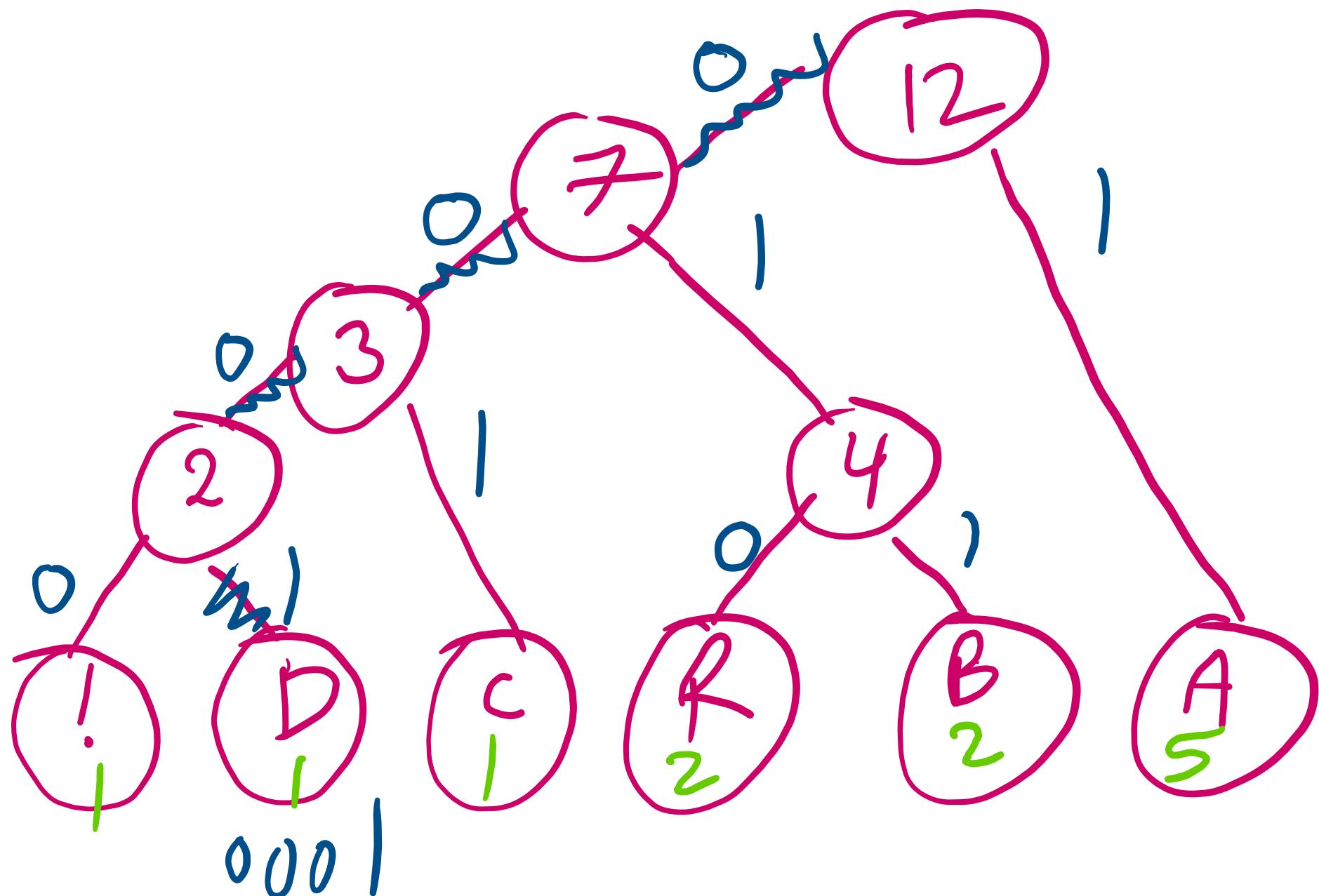
Example

- Build a tree for “ABRACADABRA!”



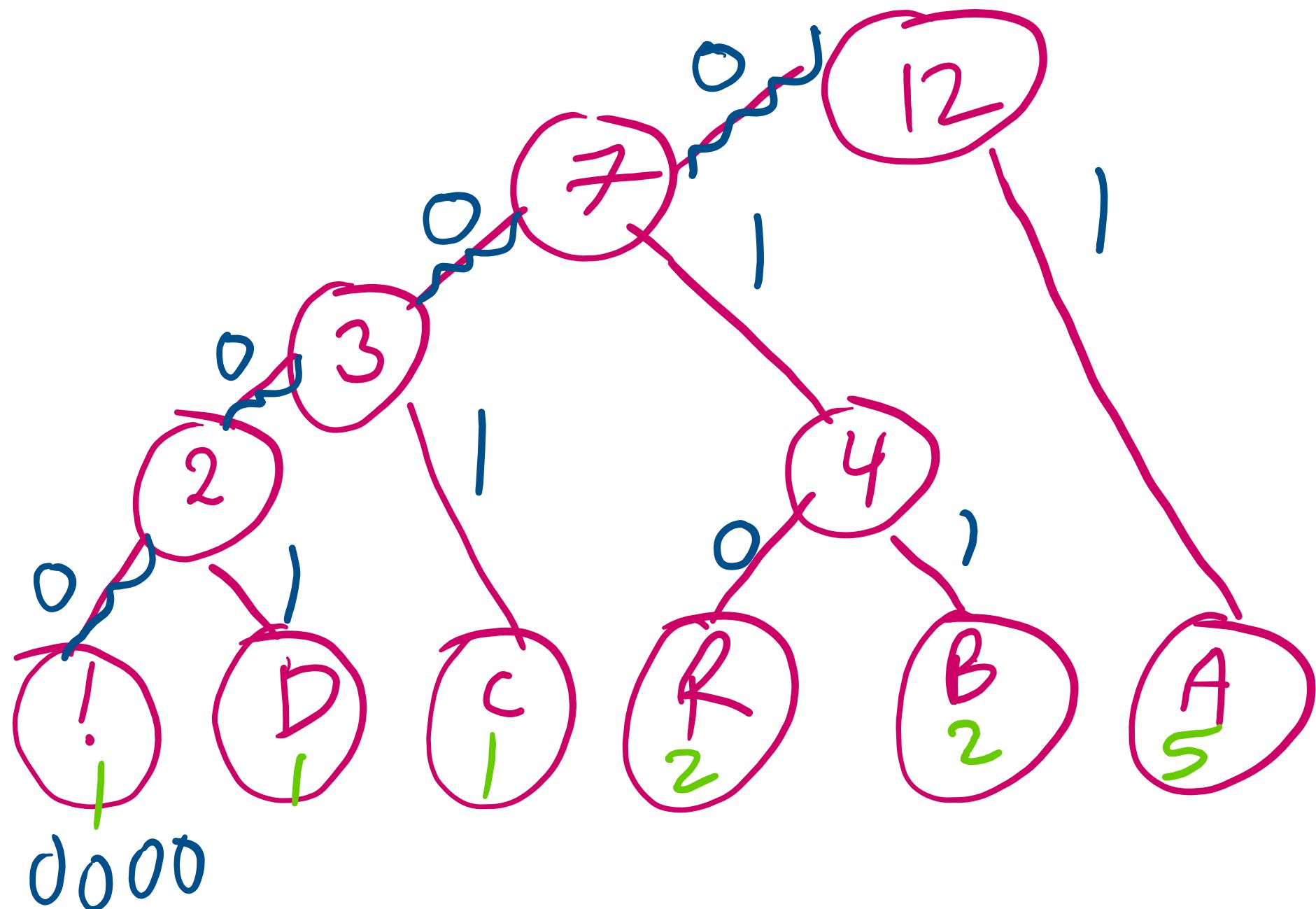
Example

- Build a tree for “ABRACADABRA!”



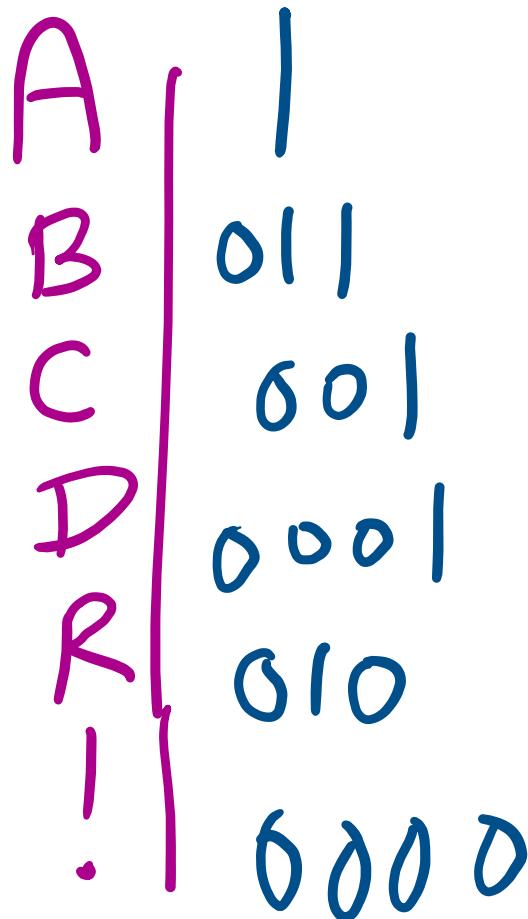
Example

- Build a tree for “ABRACADABRA!”



Example

- Build a tree for “ABRACADABRA!”



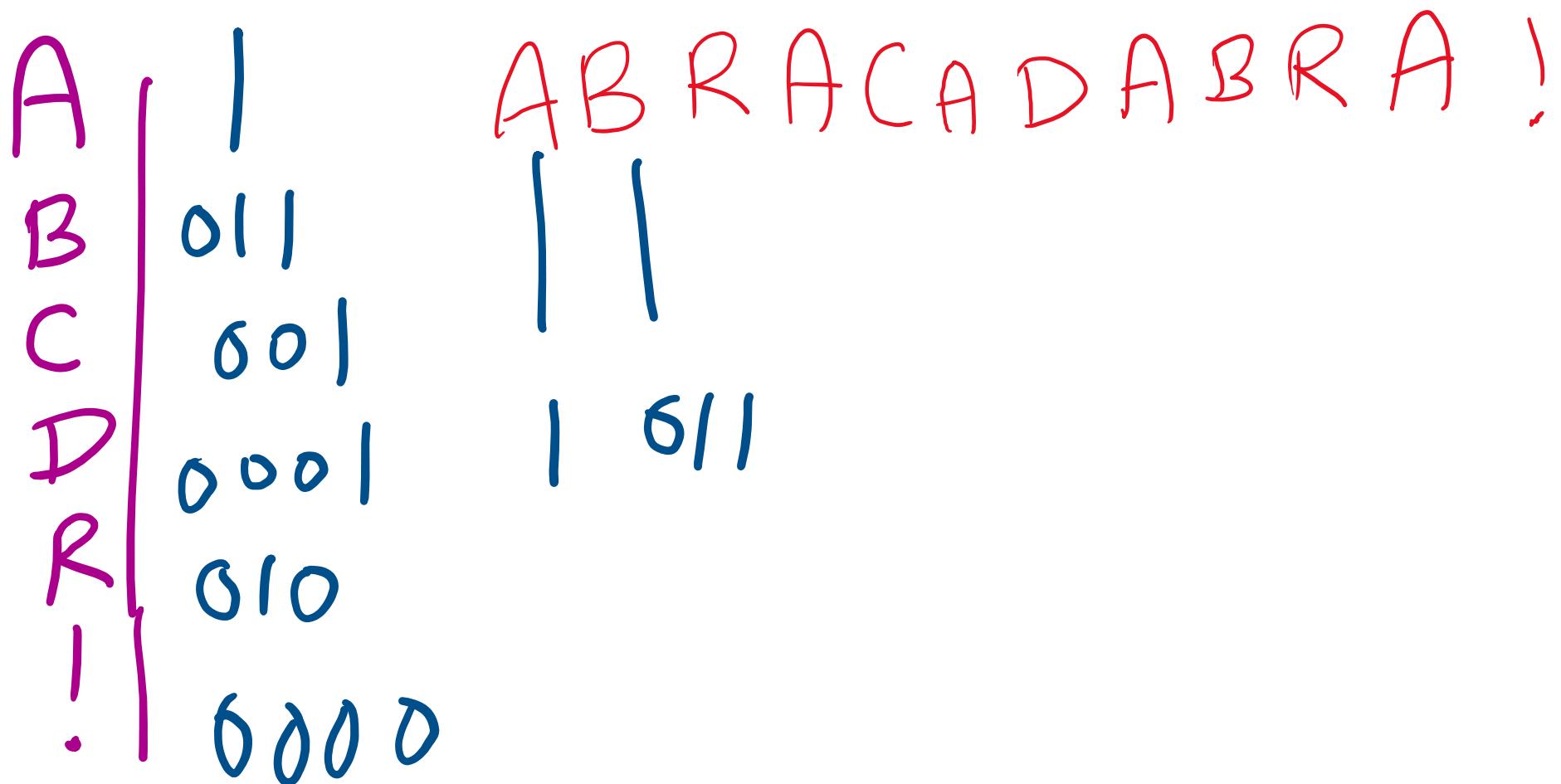
Example

- Build a tree for “ABRACADABRA!”



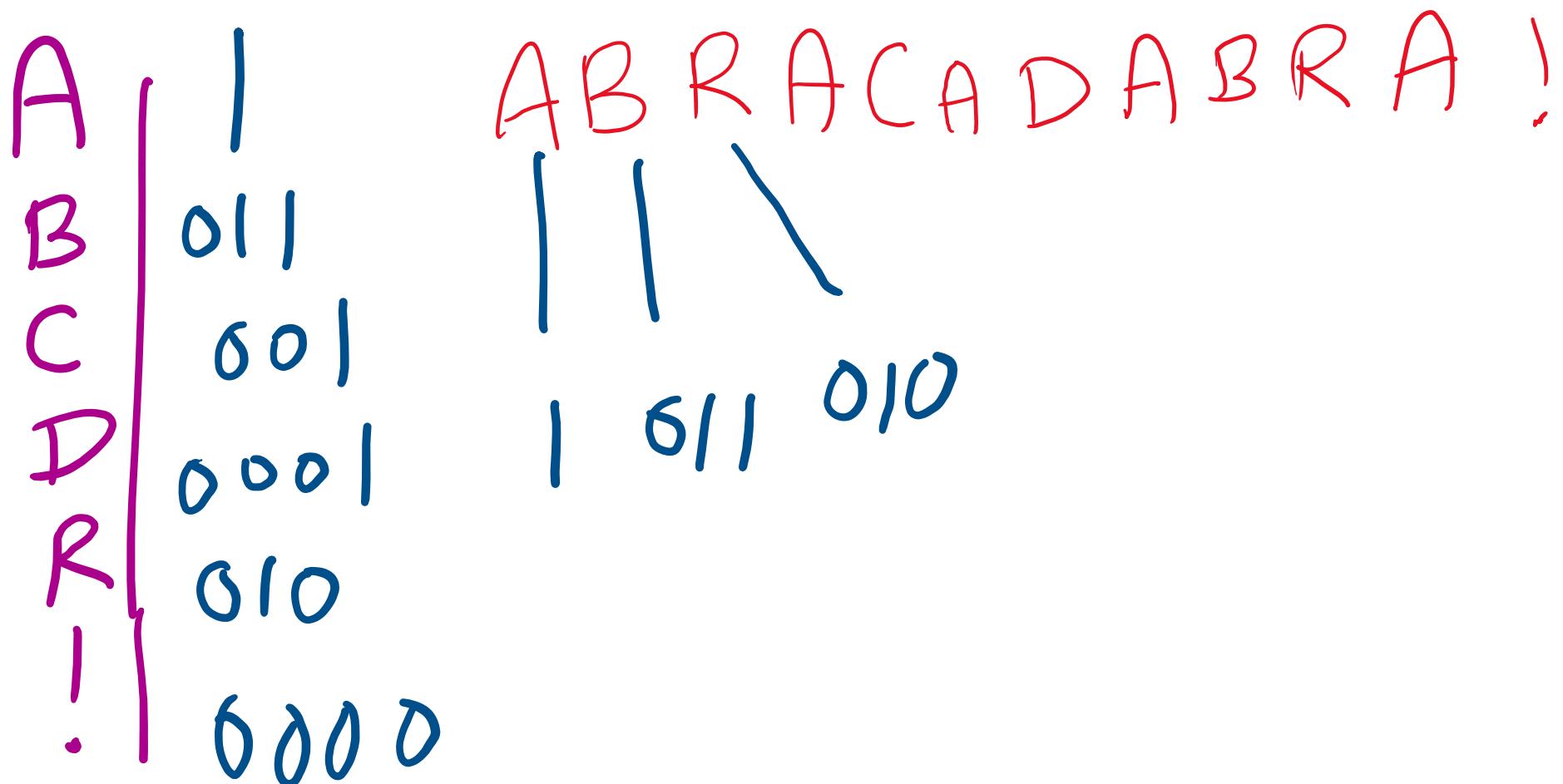
Example

- Build a tree for “ABRACADABRA!”



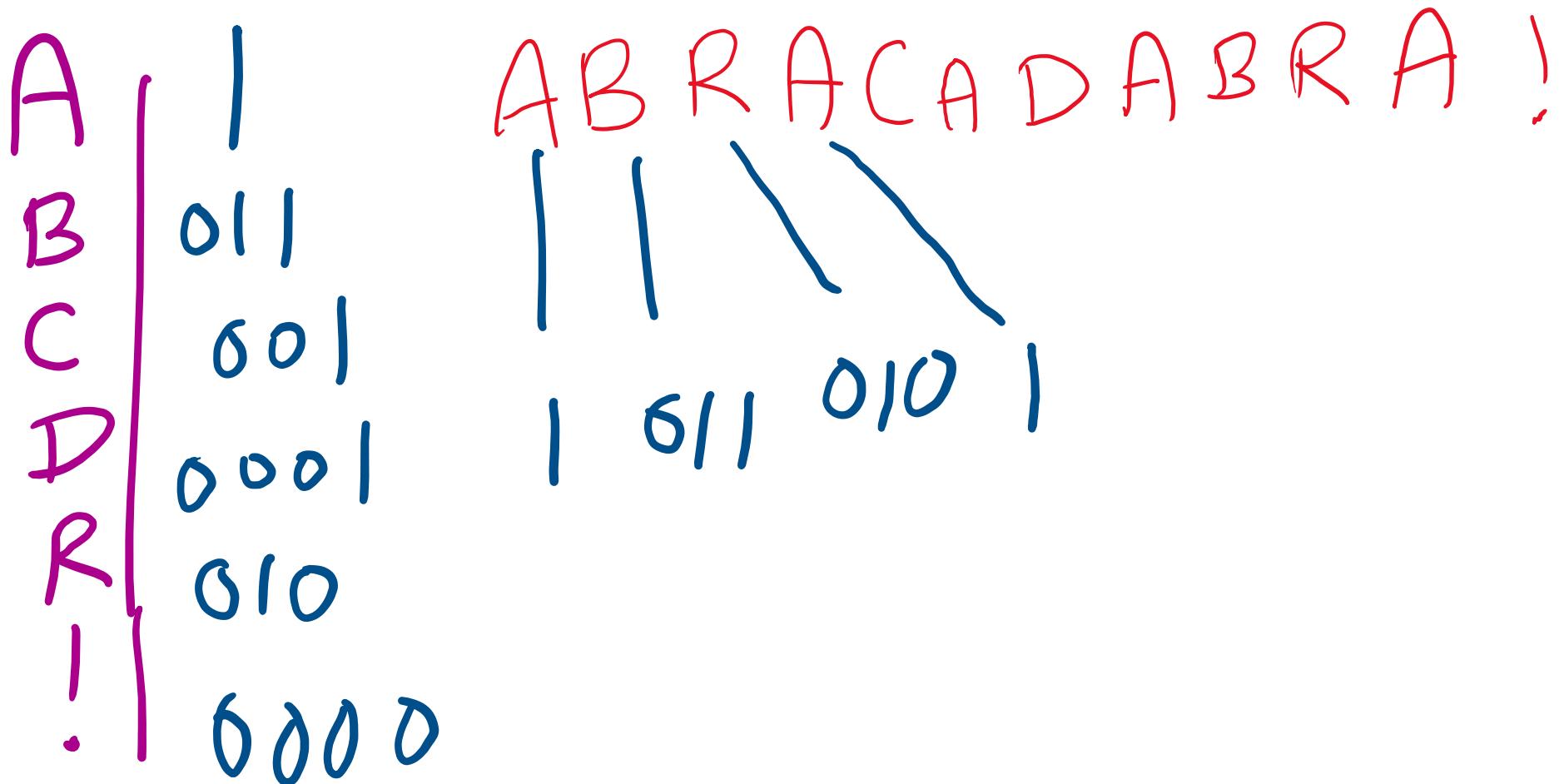
Example

- Build a tree for “ABRACADABRA!”



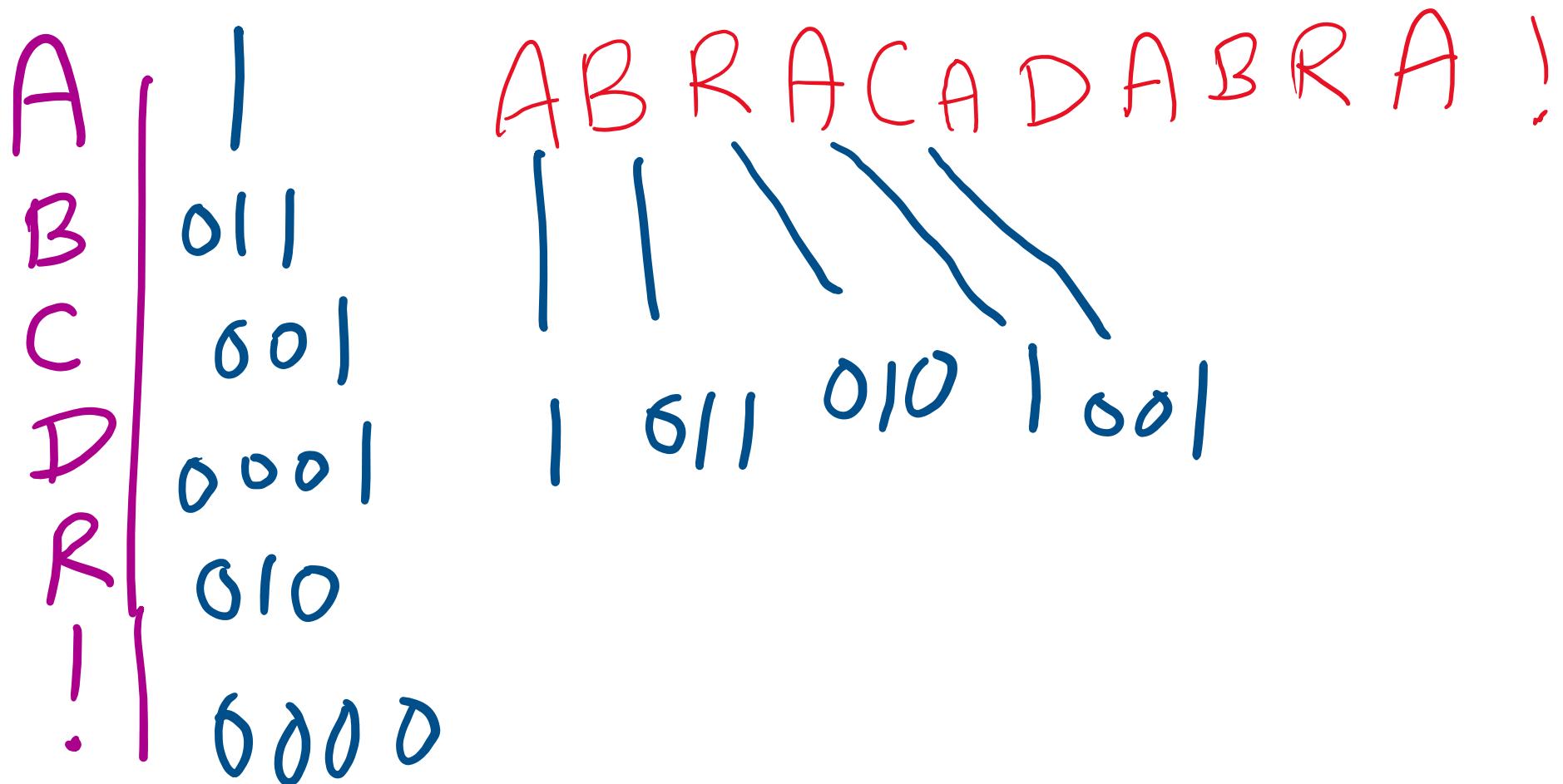
Example

- Build a tree for “ABRACADABRA!”



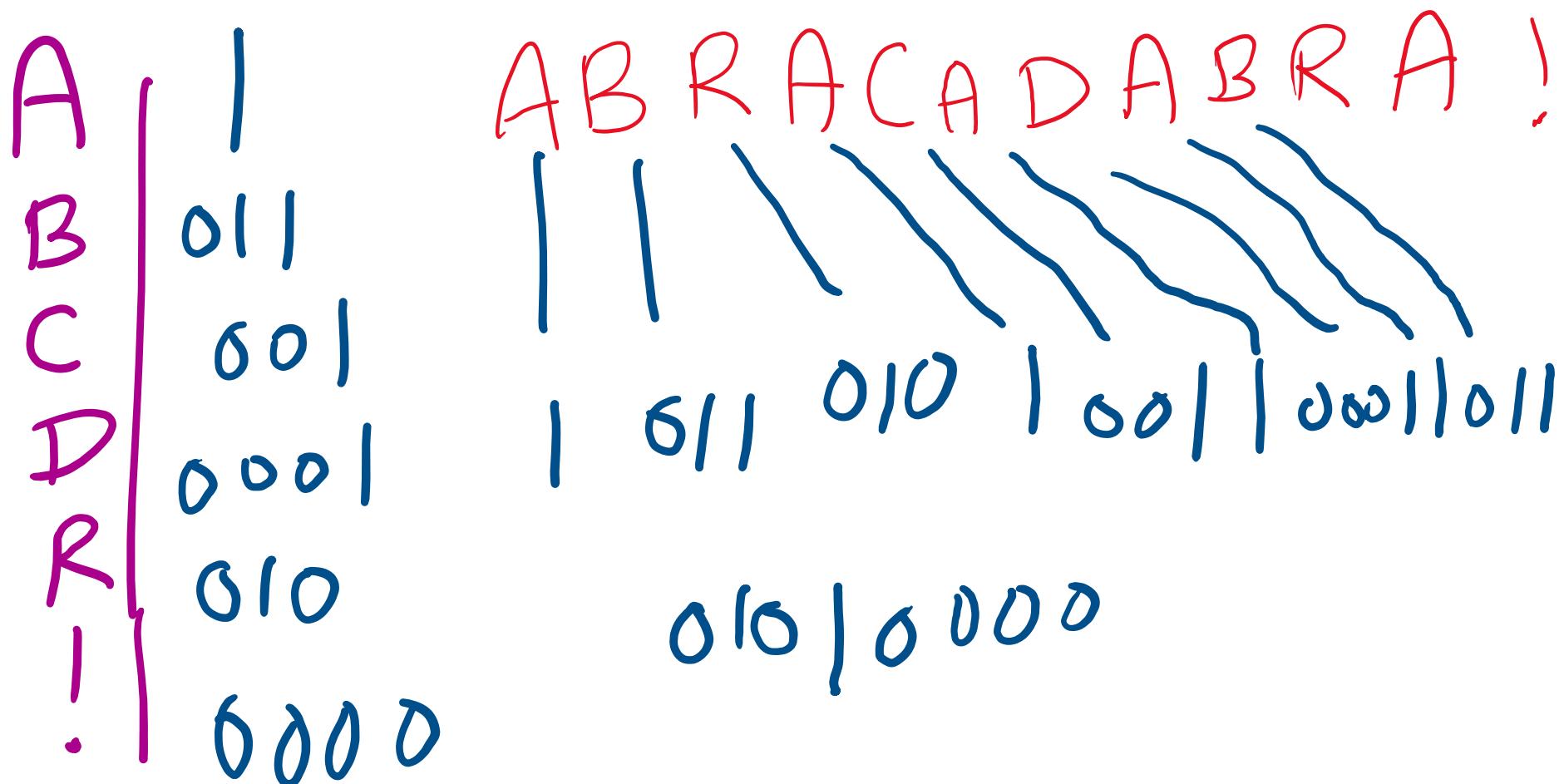
Example

- Build a tree for “ABRACADABRA!”



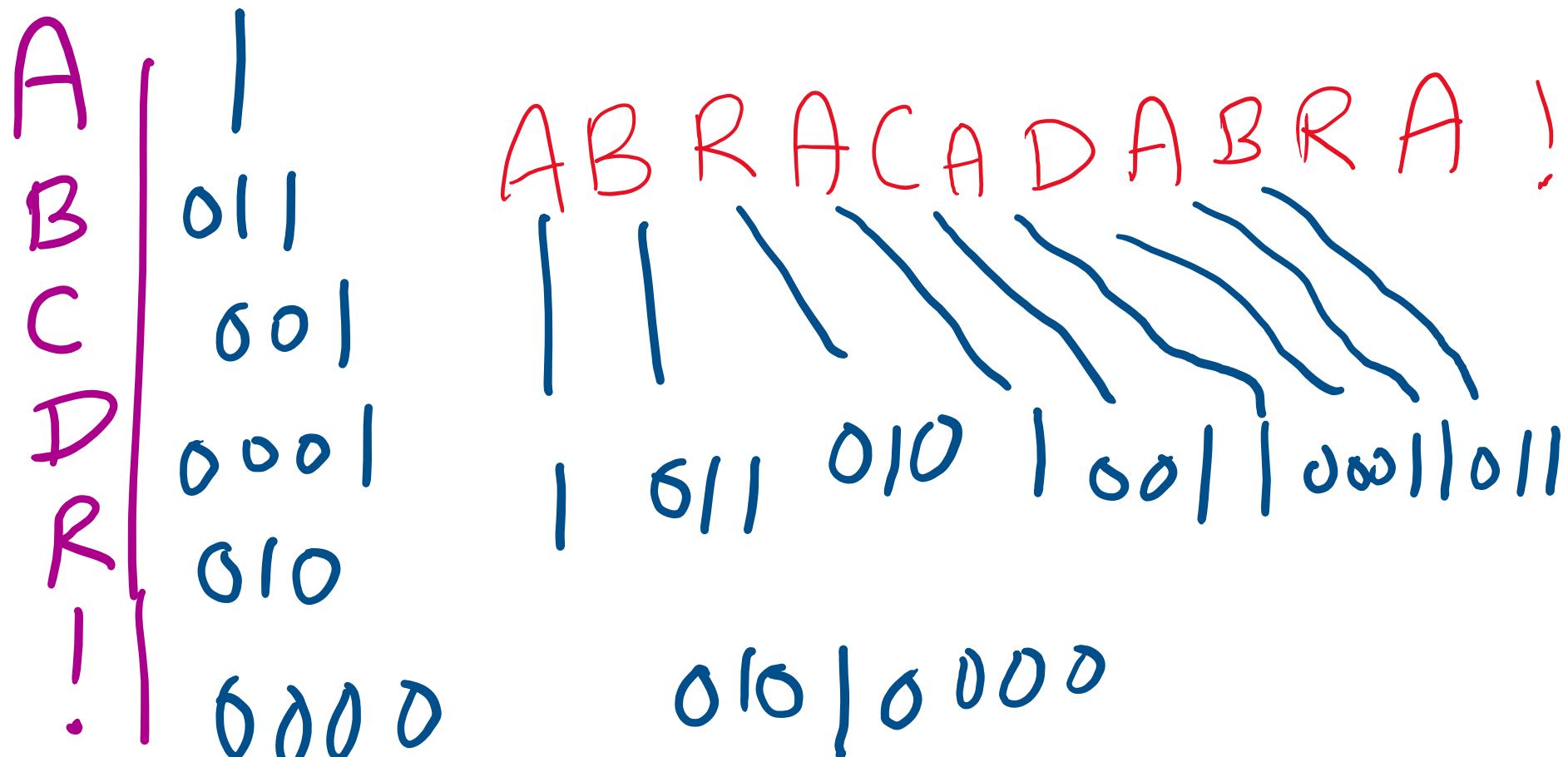
Example

- Build a tree for “ABRACADABRA!”



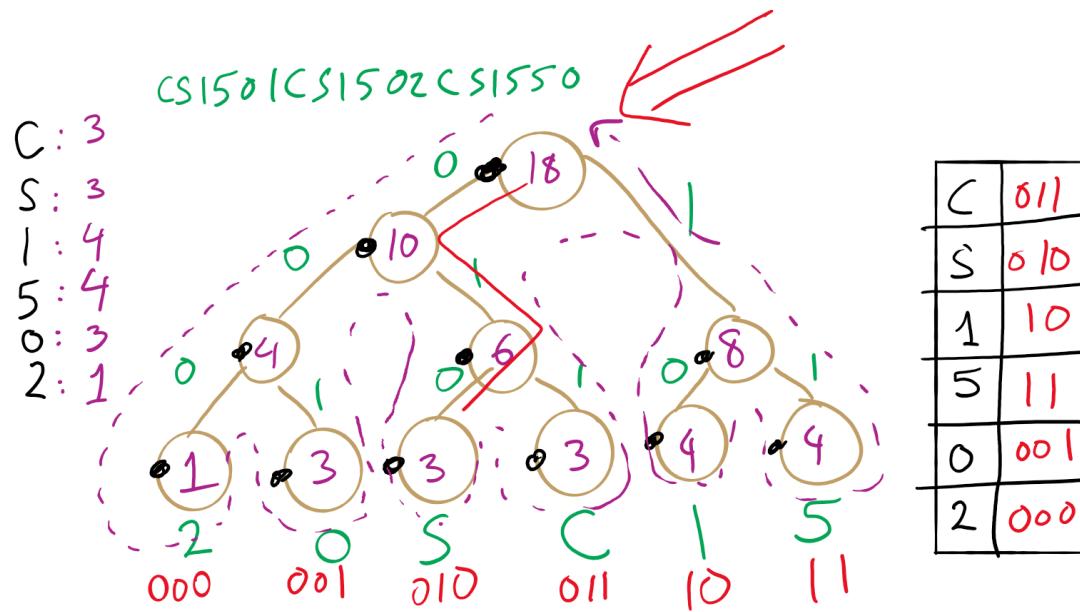
Example

- Build a tree for “ABRACADABRA!”



- Compressed size: 28
 - Original size: $12 \times 8 = 96$ bits
 - Compression ratio = 28/96

Huffman Compression Example



CS1501CS1502CS1550
011 010 10 11 001 10 011 10 11 001 000 011 010 10 11 11 001

Trie 0001 [ASCII for 2] 1 [ASCII for 0] 1 [ASCII for S] - - - -
CS1

Implementation concerns

- Need to efficiently be able to select lowest weight trees to merge when constructing the trie
 - Can accomplish this using a *priority queue*
- Need to be able to read/write bitstrings!
 - Unless we pick multiples of 8 bits for our codewords, we will need to read/write *fractions* of bytes for our codewords
 - We're not actually going to do I/O on fraction of bytes
 - We'll maintain a buffer of bytes and perform bit processing on this buffer
 - See `BinaryStdIn.java` and `BinaryStdOut.java`

We need to read and write individual bits: Binary I/O

```
private static void writeBit(boolean bit) {  
    // add bit to buffer by shifting in a zero  
    buffer <<= 1;  
    if (bit) buffer |= 1; //then turning it to one if needed  
    // if buffer is full (8 bits), write out as a single byte  
    N++;  
    if (N == 8) clearBuffer();  
}
```

```
writeBit(true);  
writeBit(false);  
writeBit(true);  
writeBit(false);  
writeBit(false);  
writeBit(false);  
writeBit(false);  
writeBit(true);
```

buffer:

00000000

N:

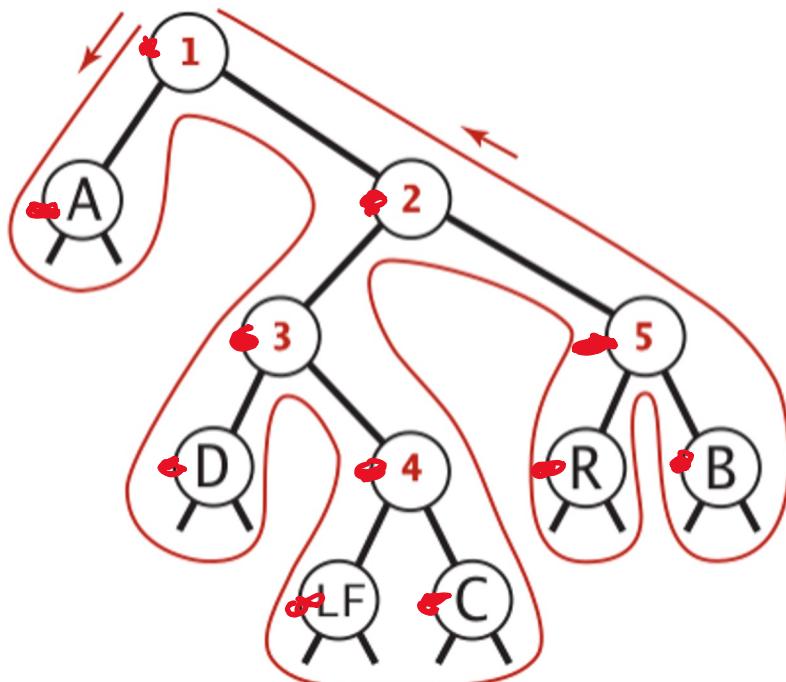
0

Further implementation concerns

- To encode, we'll need to read in characters and output codes
- To decode, we'll need to read in codes and output characters
- Sounds like we'll need a symbol table!
 - What implementation would be best?
 - Same for encoding and decoding?
 - Note that this means we need access to the trie to expand a compressed file!
 - Let's store the trie in the compressed file
 - how?

Representing tries as bitstrings

Preorder traversal

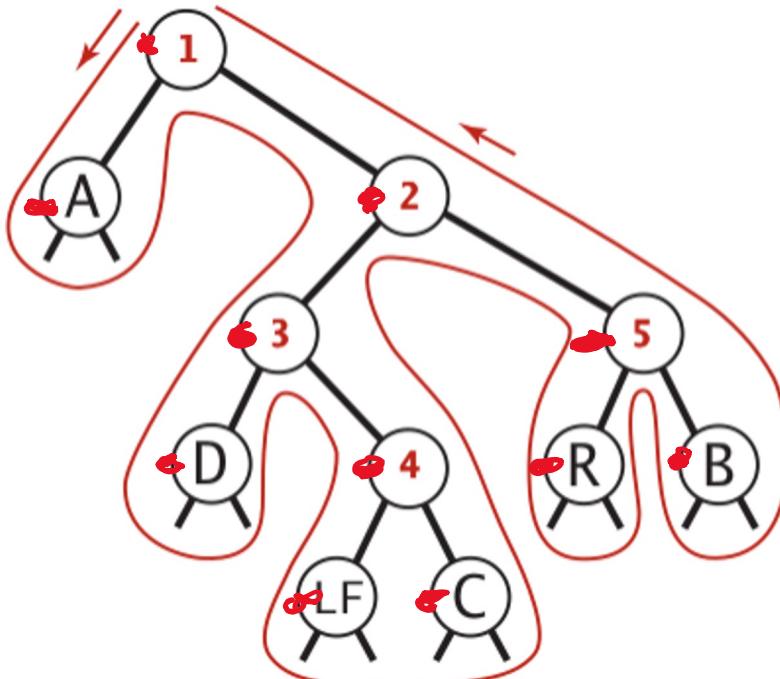


internal node → 0

leaf node → 1 followed by ASCII code of char inside

Representing tries as bitstrings

Preorder traversal

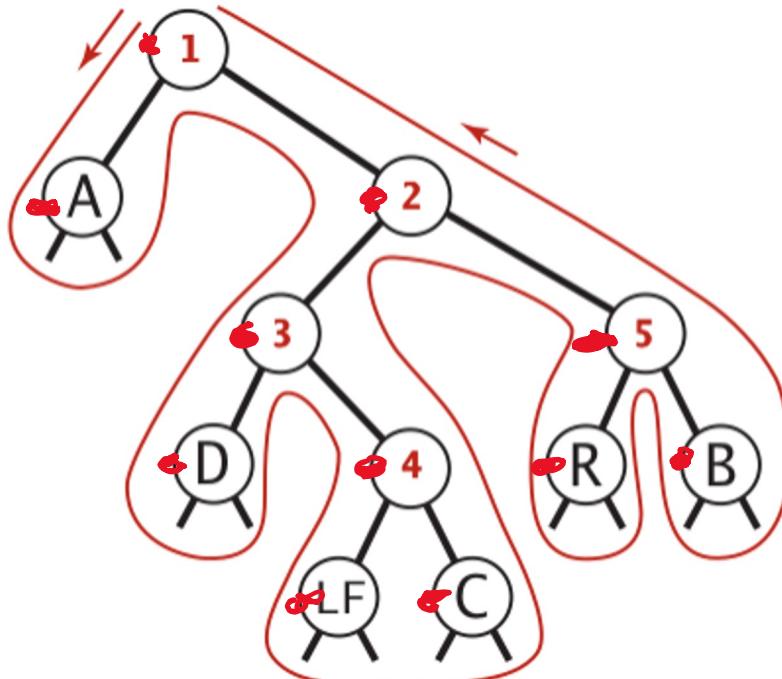


leaf

0
↑
1

Representing tries as bitstrings

Preorder traversal

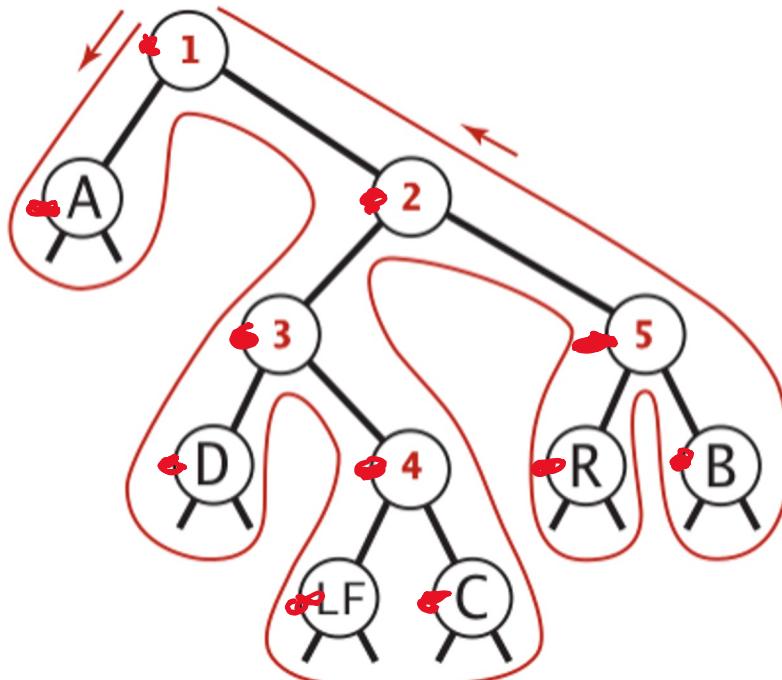


leaves

↓
A
0101000001
↑
1

Representing tries as bitstrings

Preorder traversal

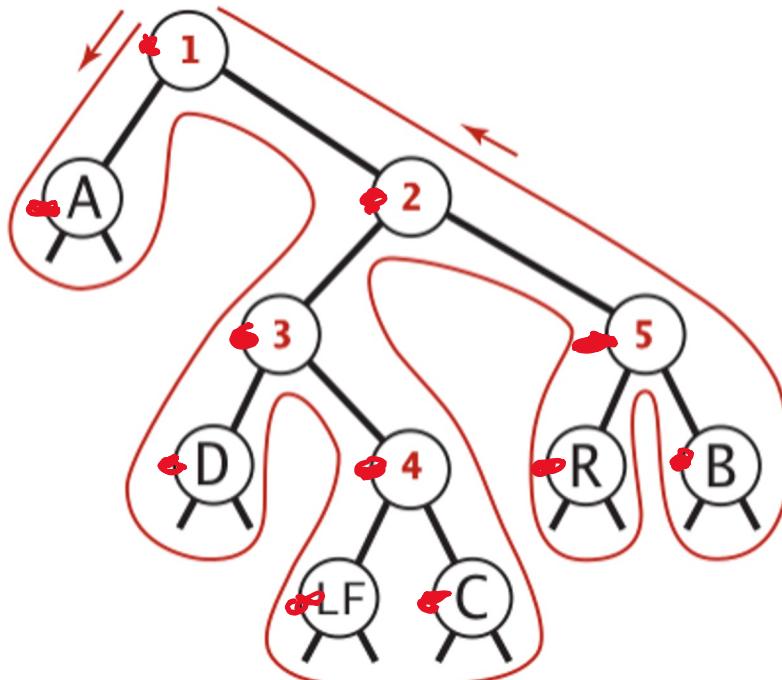


leaves

↓
A
01010000010
↑ 1 ↑ 2

Representing tries as bitstrings

Preorder traversal

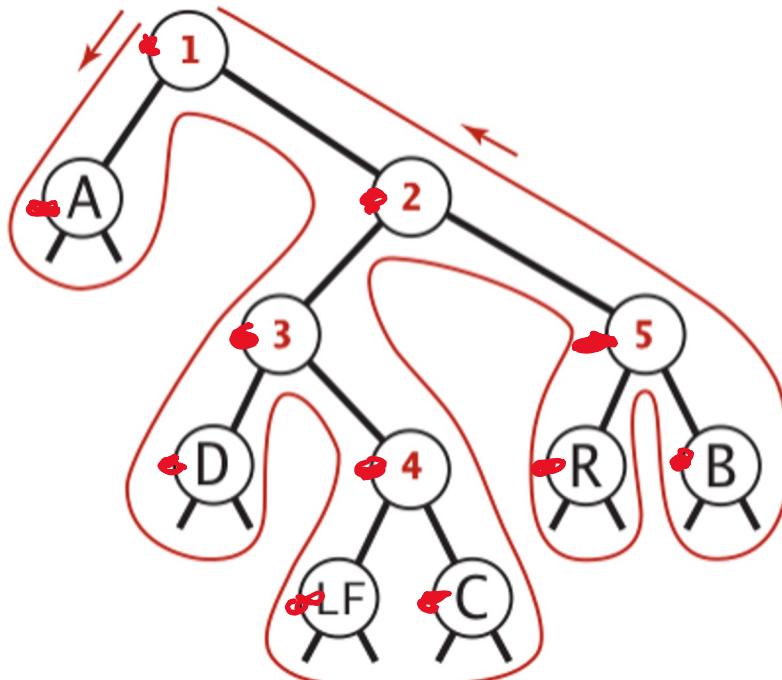


leaves

↓
A
010100000100:
↑
1 2 3

Representing tries as bitstrings

Preorder traversal

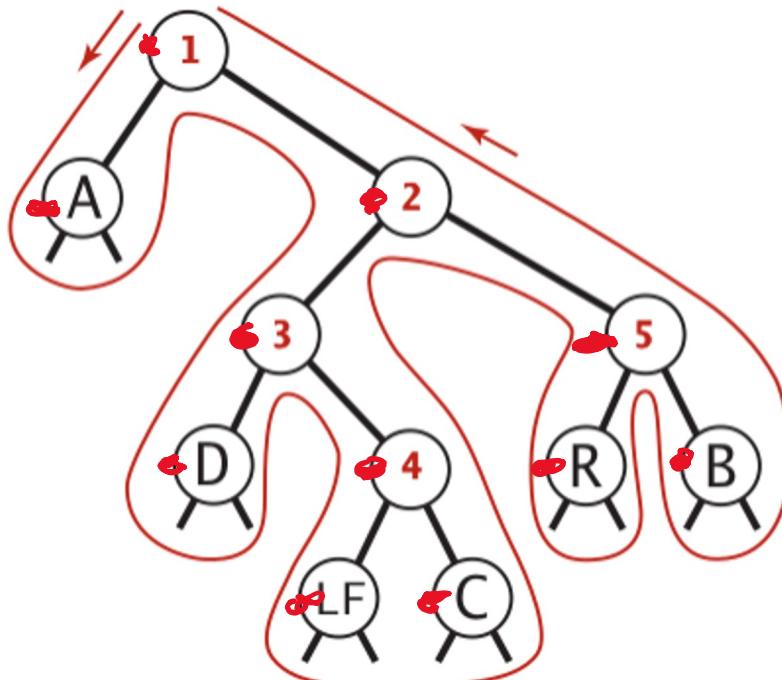


leaves

↓ A ↓ D
010100000100101000100
↑ ↑
1 2 3

Representing tries as bitstrings

Preorder traversal

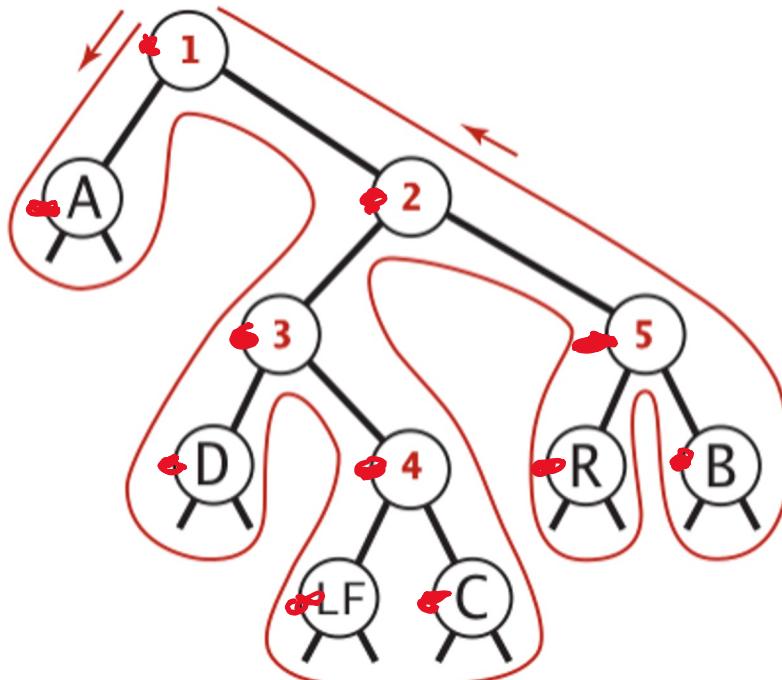


leaves

↓	A	↓	D	
<hr/>				
0	101000001	00	101000100	0
↑		↑↑		↑
1		2 3		4

Representing tries as bitstrings

Preorder traversal

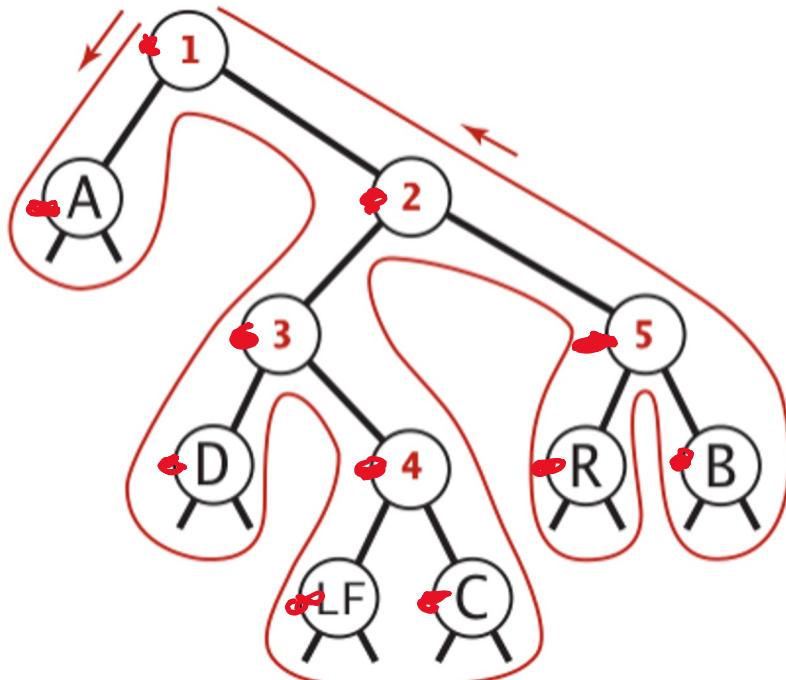


leaves

↓	<u>A</u>	↓	<u>D</u>	↓	<u>LF</u>
0	101000001	001	01010001000	0	1000010101
↑		↑↑		↑	
1		2 3		4	

Representing tries as bitstrings

Preorder traversal

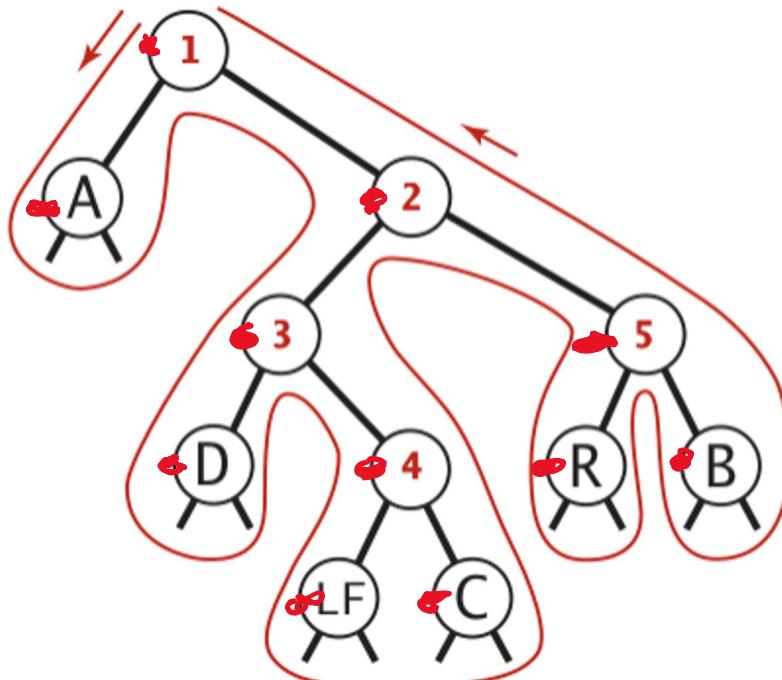


leaves

↓	<u>A</u>	↓	<u>D</u>	↓	<u>LF</u>	↓	<u>C</u>
0	101000001	001	01010001000	0	1000010101010101000011		
↑		↑↑		↑			
1		2 3		4			

Representing tries as bitstrings

Preorder traversal

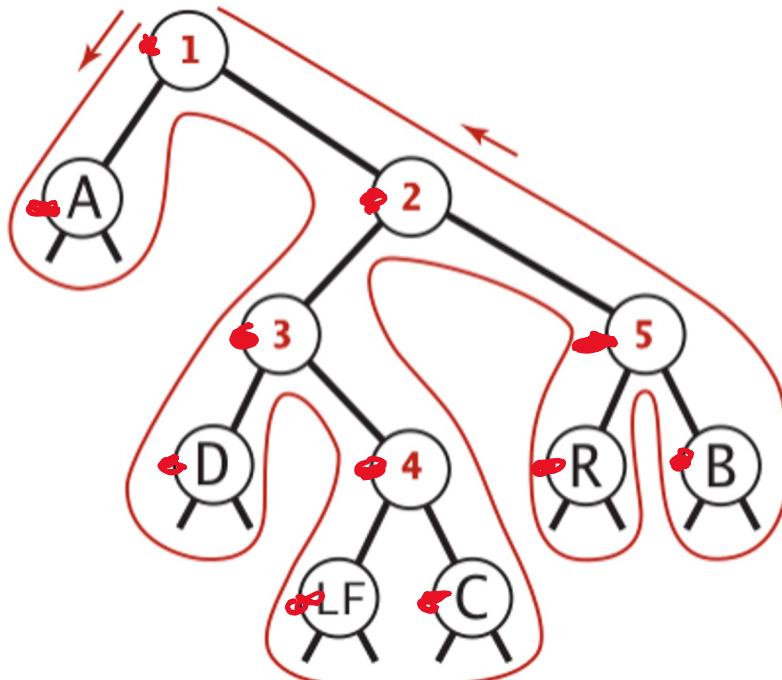


leaves

↓	<u>A</u>	↓	<u>D</u>	↓	<u>LF</u>	↓	<u>C</u>	↓
0	101000001	00	1010001000	0	100001010101010000110	0		
↑		↑↑		↑			↑	
1		2 3		4			5	

Representing tries as bitstrings

Preorder traversal

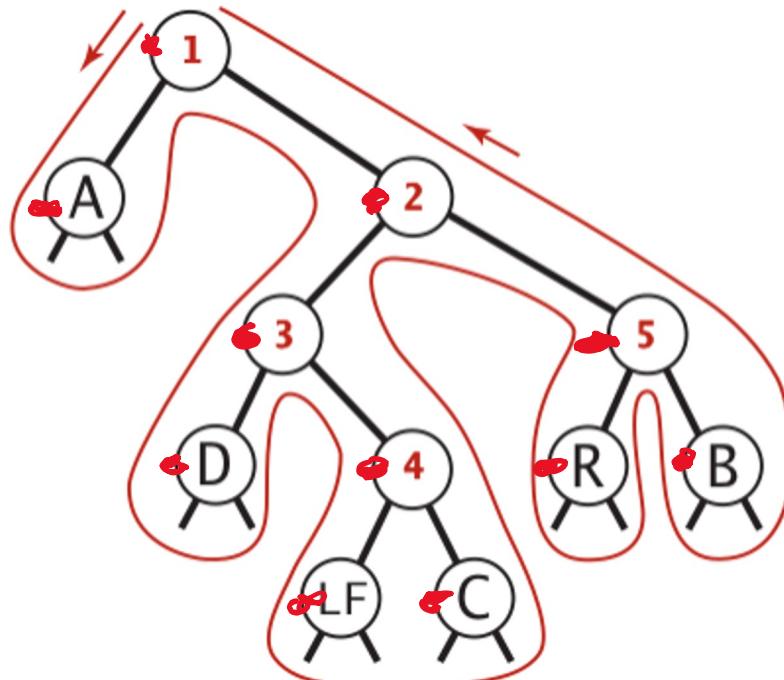


leaves

↓	<u>A</u>	↓	<u>D</u>	↓	<u>LF</u>	↓	<u>C</u>	↓	<u>R</u>
0	101000001	00	1010001000	0	1000010101	0101000011	0101010010		
↑ 1		↑ 2 3		↑ 4			↑ 5	← int	

Representing tries as bitstrings

Preorder traversal



leaves

	A		D		LF		C		R		B
↓	0101000001	00	↓	01010001000	0	↓	10000101010101000011	0	10101001010101000010	↓	
↑	1		↑↑	2 3		↑	4		↑	5	← internal nodes

Writing a trie

```
private static void writeTrie(Node x){  
    if (x.isLeaf()) {  
        BinaryStdOut.write(true);  
        BinaryStdOut.write(x.ch);  
        return;  
    }  
    BinaryStdOut.write(false);  
    writeTrie(x.left);  
    writeTrie(x.right);  
}
```

Reading a trie back

Similar to the read tree lab!

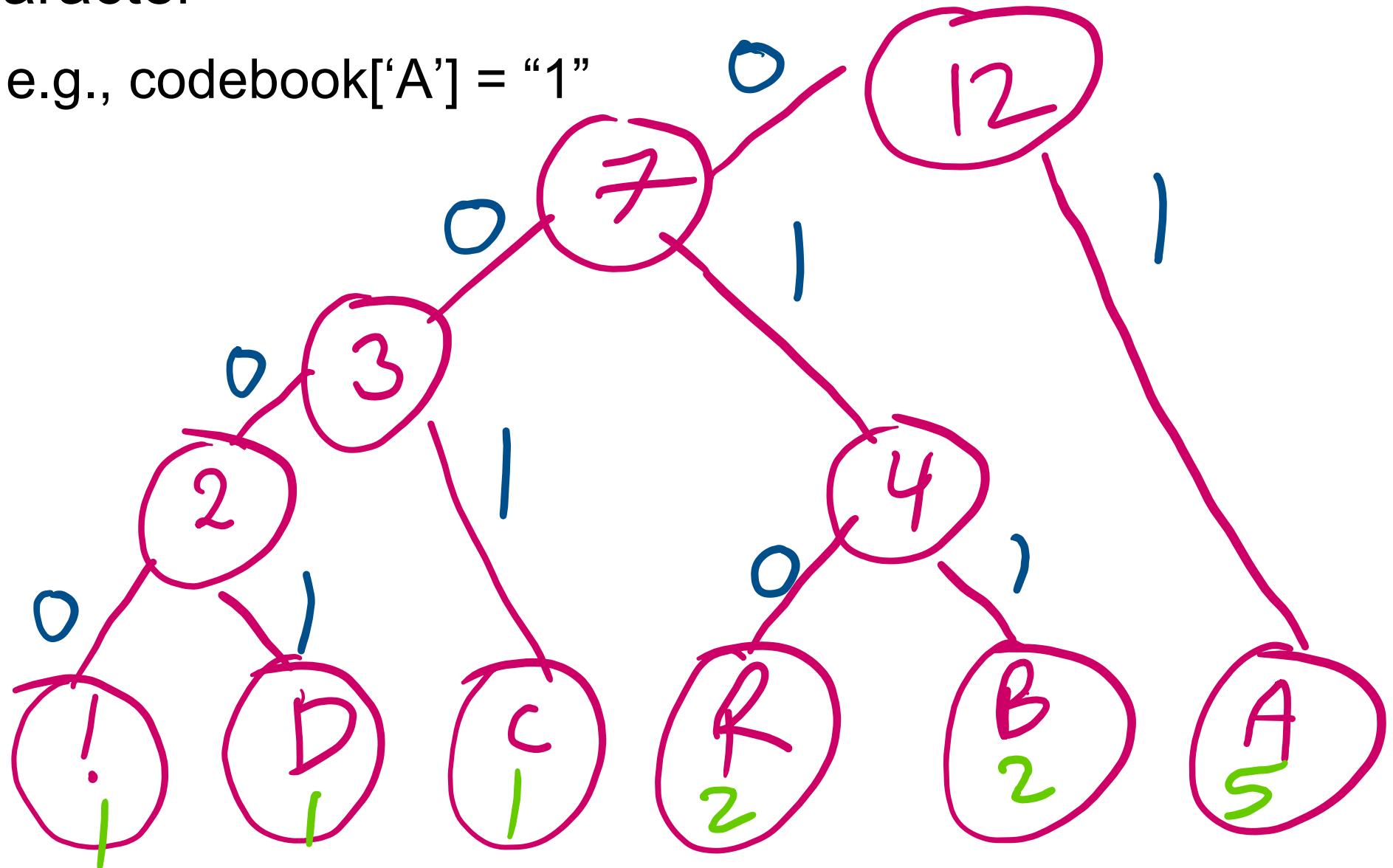
```
private static Node readTrie() {  
    if (BinaryStdIn.readBoolean())  
        return new  
Node(BinaryStdIn.readChar(), 0, null,  
null);  
  
    return new Node('\0', 0, readTrie(),  
readTrie());  
}
```

Huffman pseudocode

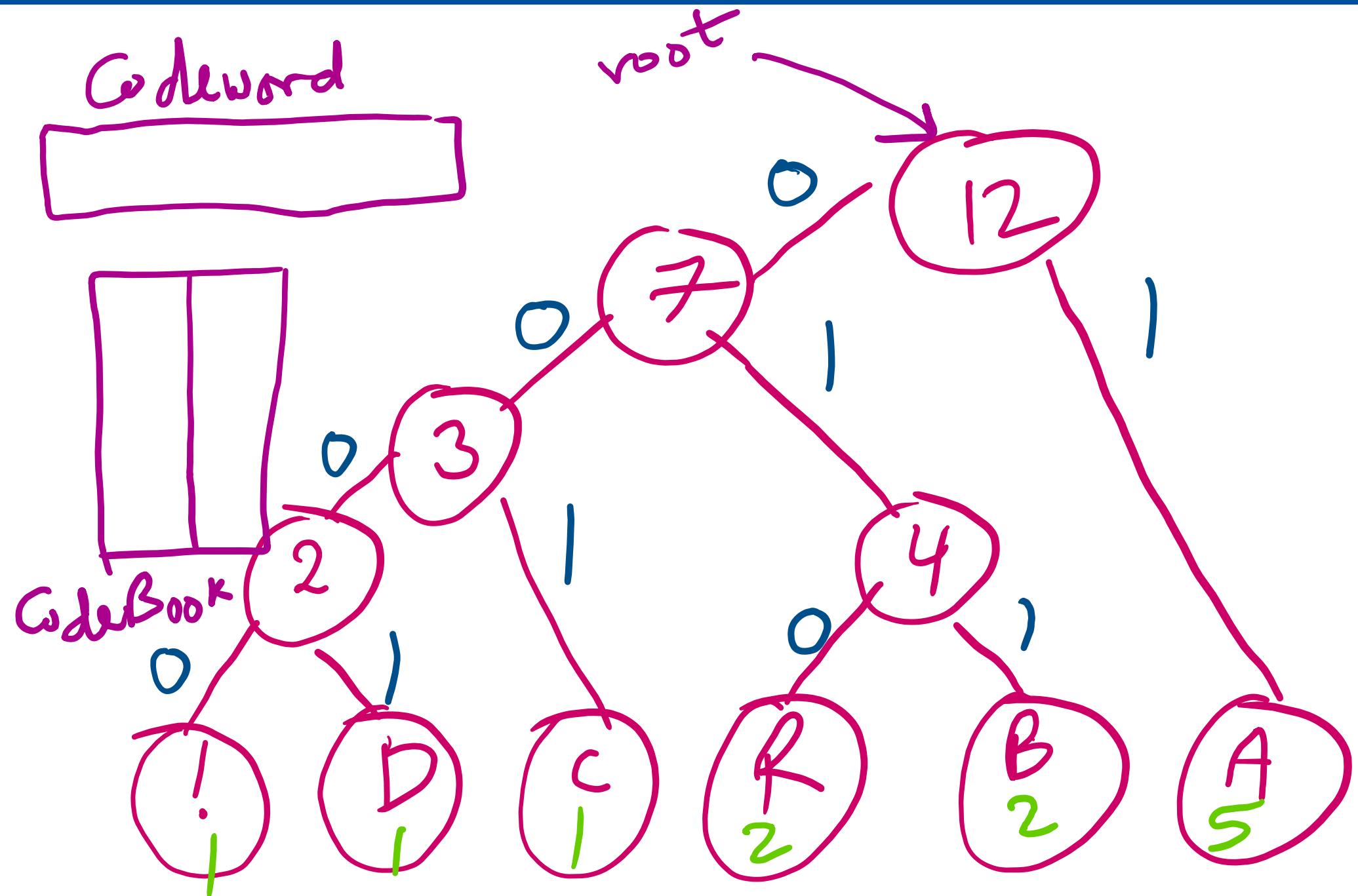
- Encoding approach:
 - Read input
 - Compute frequencies
 - Build trie/codeword table
 - Write out trie as a bitstring to compressed file
 - Write out character count of input (**why is that necessary?**)
 - Use table to write out the codeword for each input character
- Decoding approach:
 - Read trie
 - Read character count
 - Use trie to decode bitstring of compressed file

Huffman Compression: Generating the Codebook

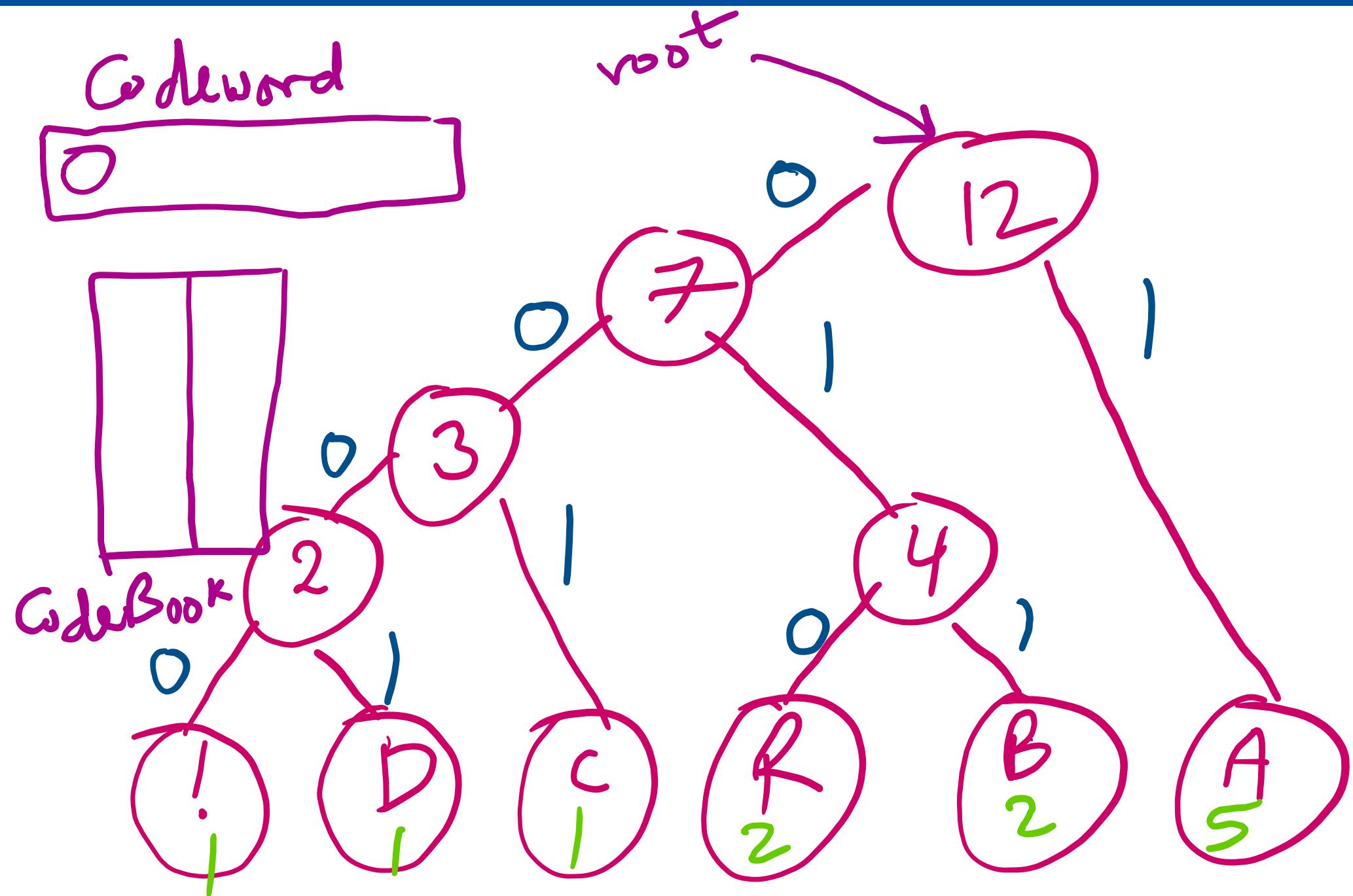
- The codebook is a table of codewords indexed by character
 - e.g., $\text{codebook}['A'] = "1"$



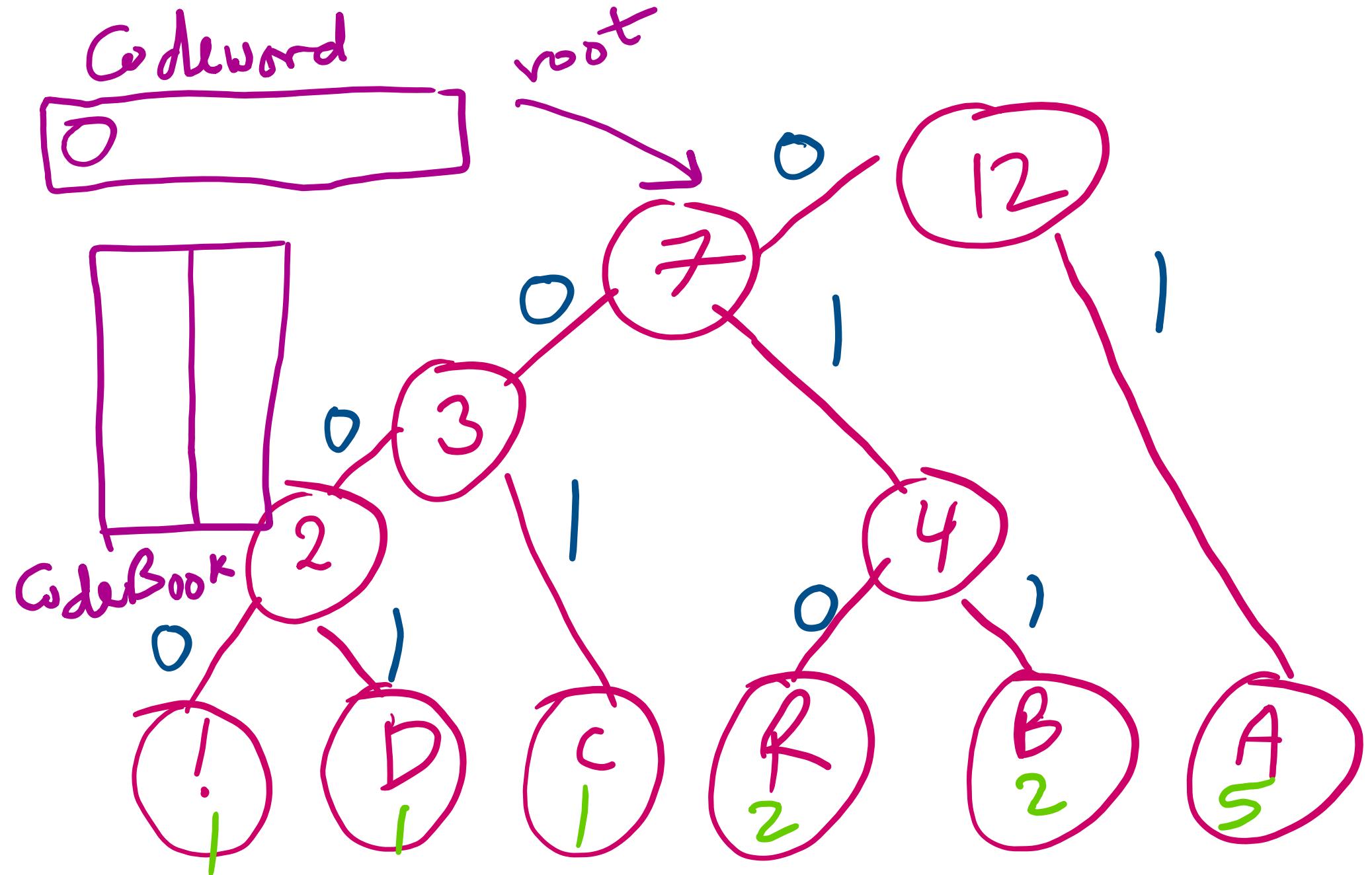
Huffman Compression: Generating the Codebook



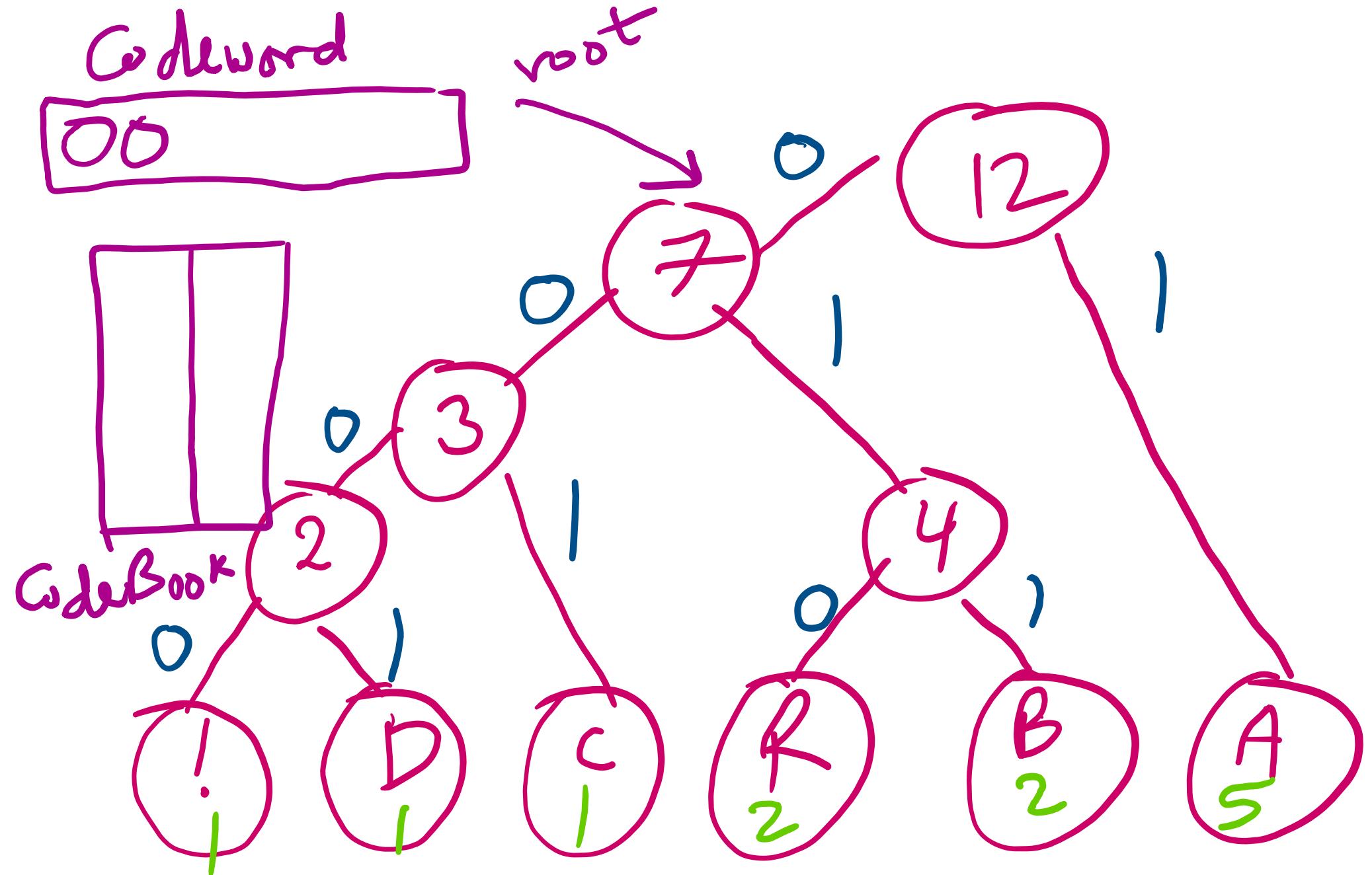
Huffman Compression: Generating the Codebook



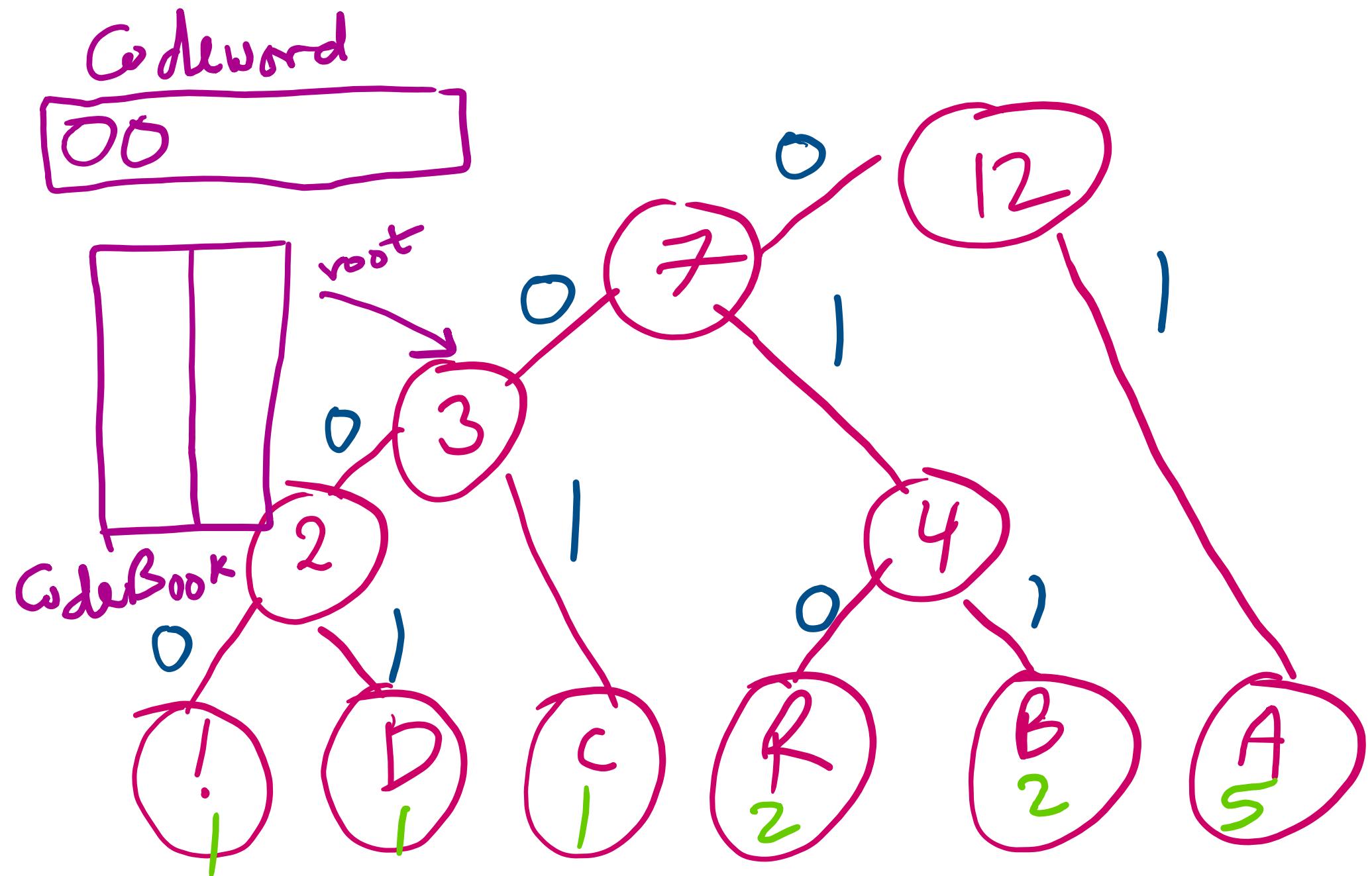
Huffman Compression: Generating the Codebook



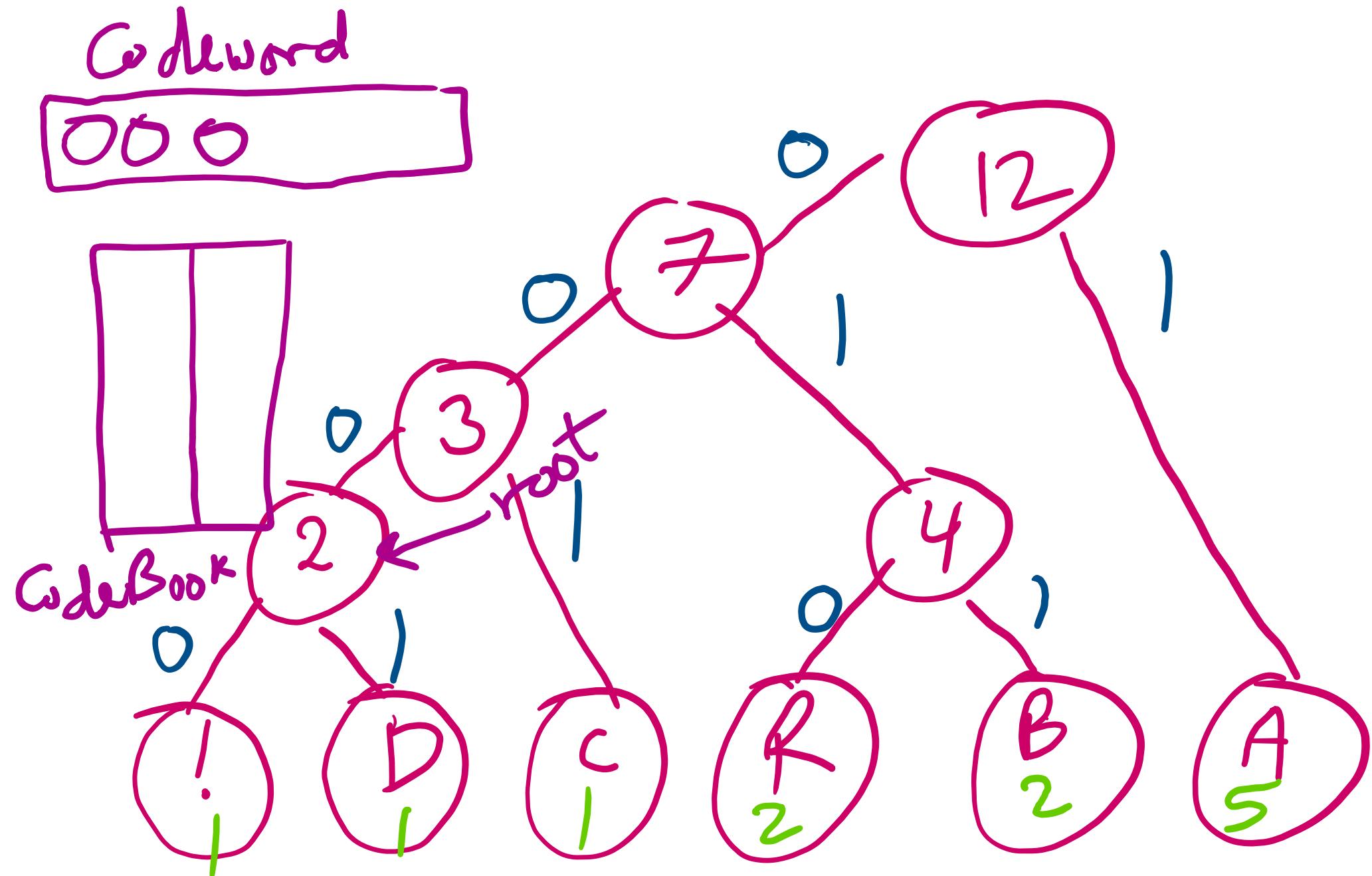
Huffman Compression: Generating the Codebook



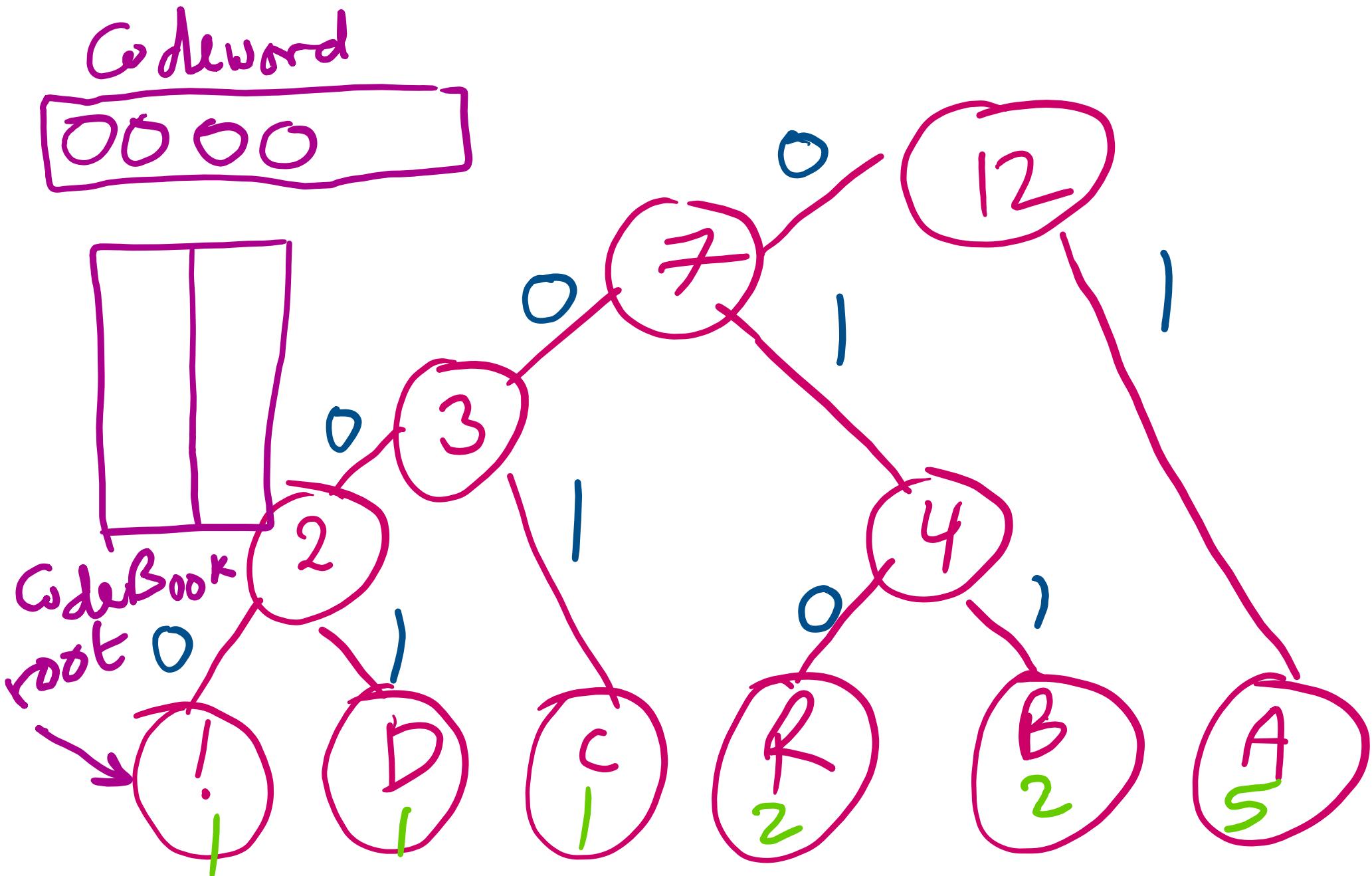
Huffman Compression: Generating the Codebook



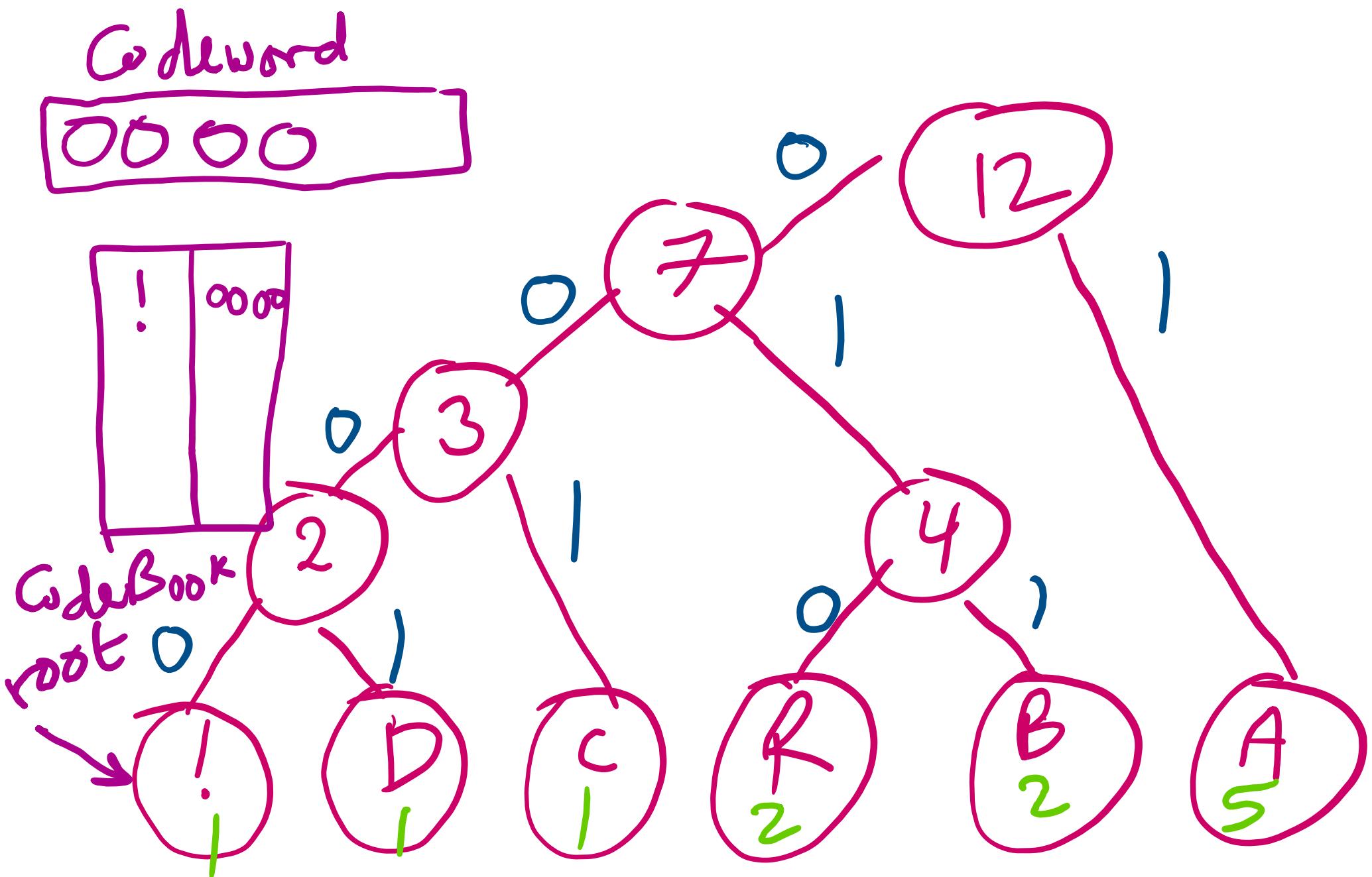
Huffman Compression: Generating the Codebook



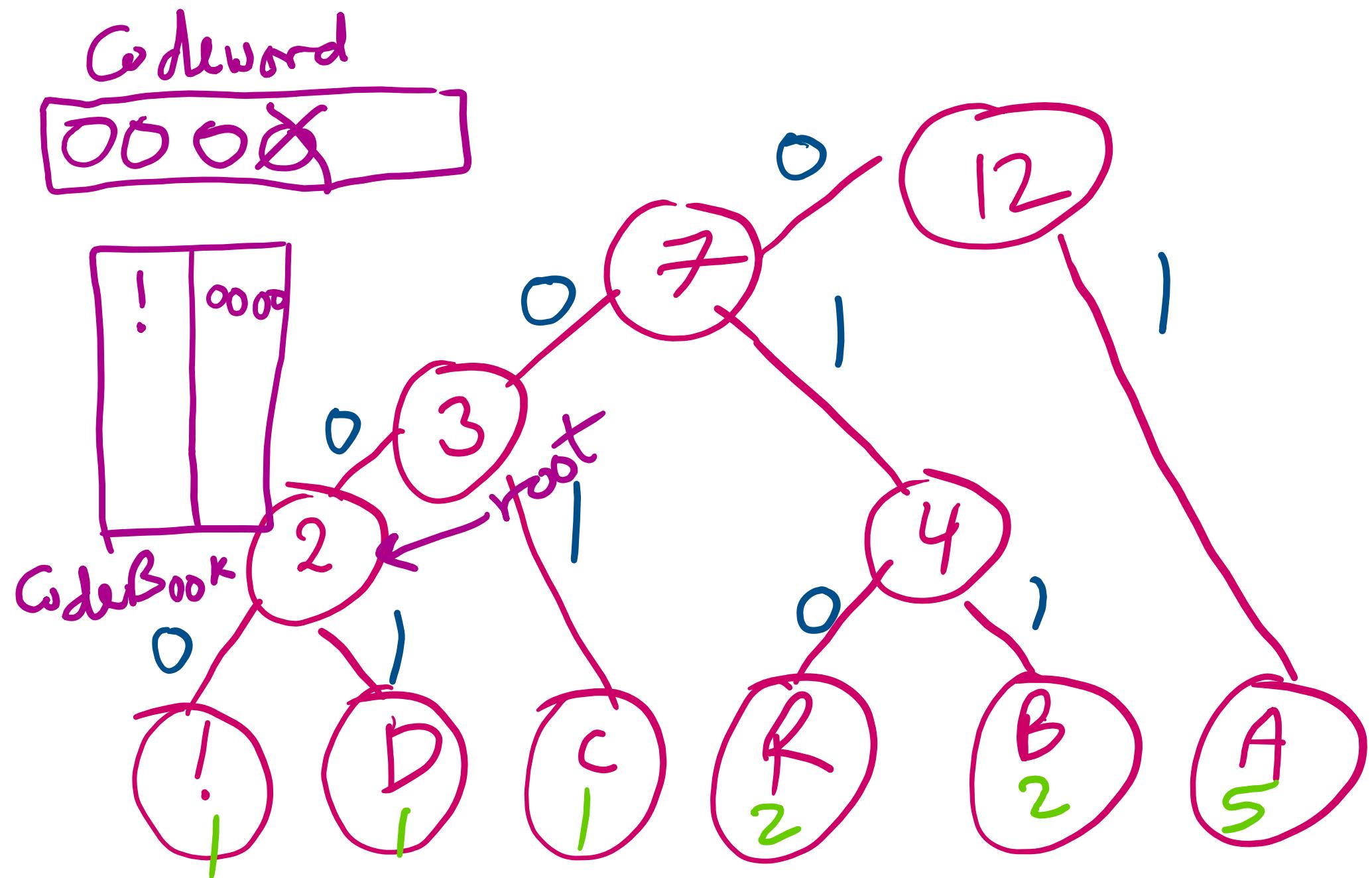
Huffman Compression: Generating the Codebook



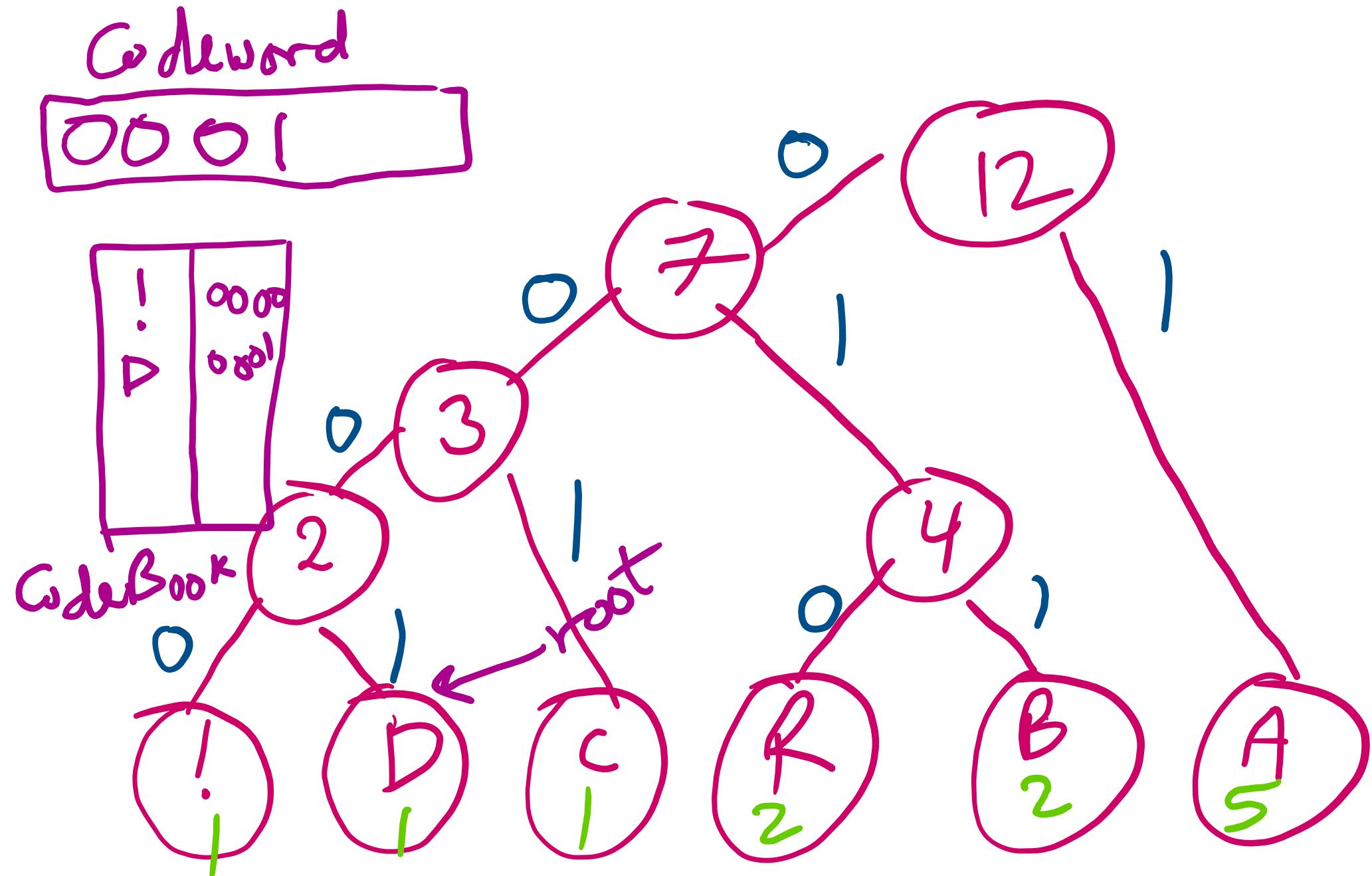
Huffman Compression: Generating the Codebook



Huffman Compression: Generating the Codebook



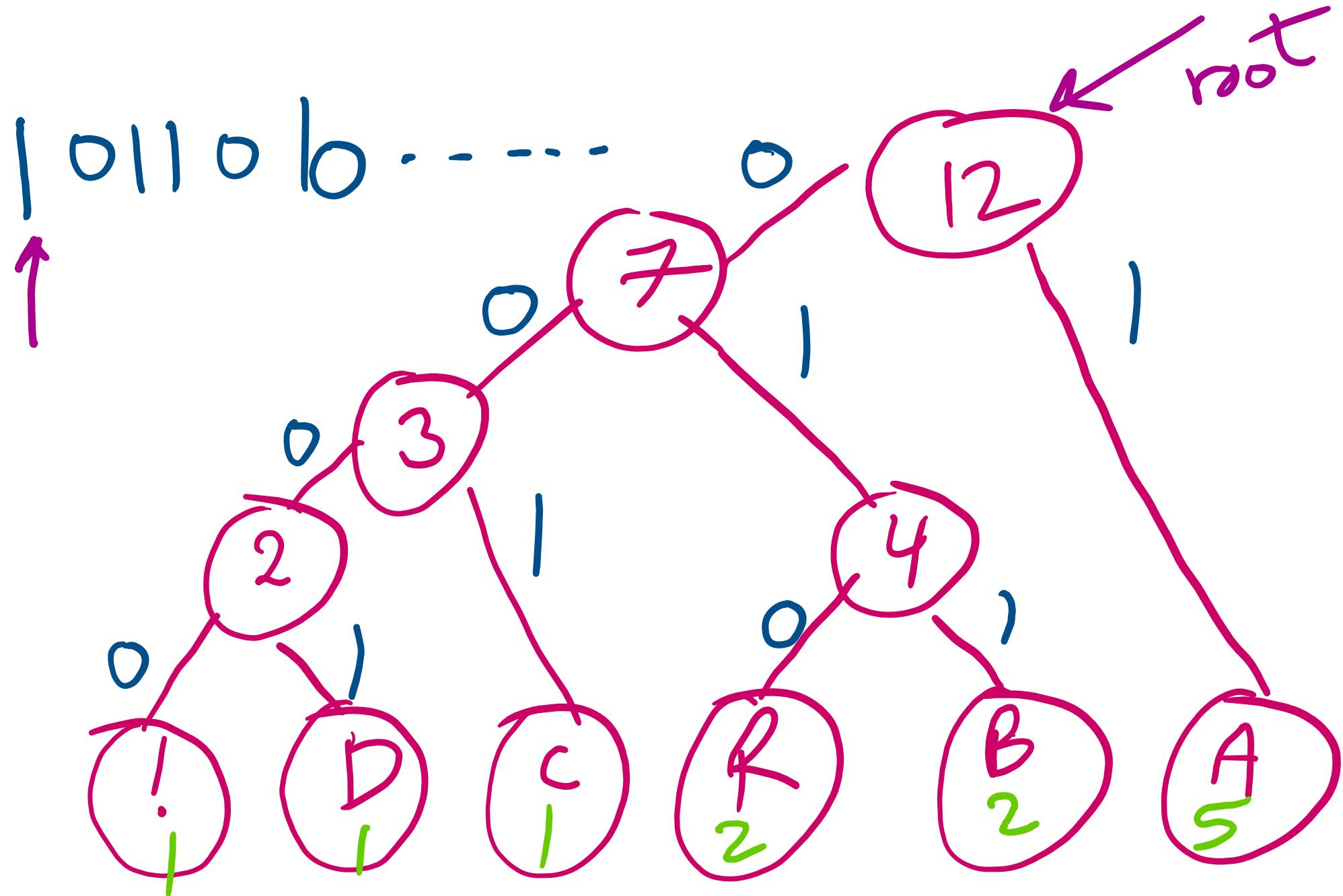
Huffman Compression: Generating the Codebook



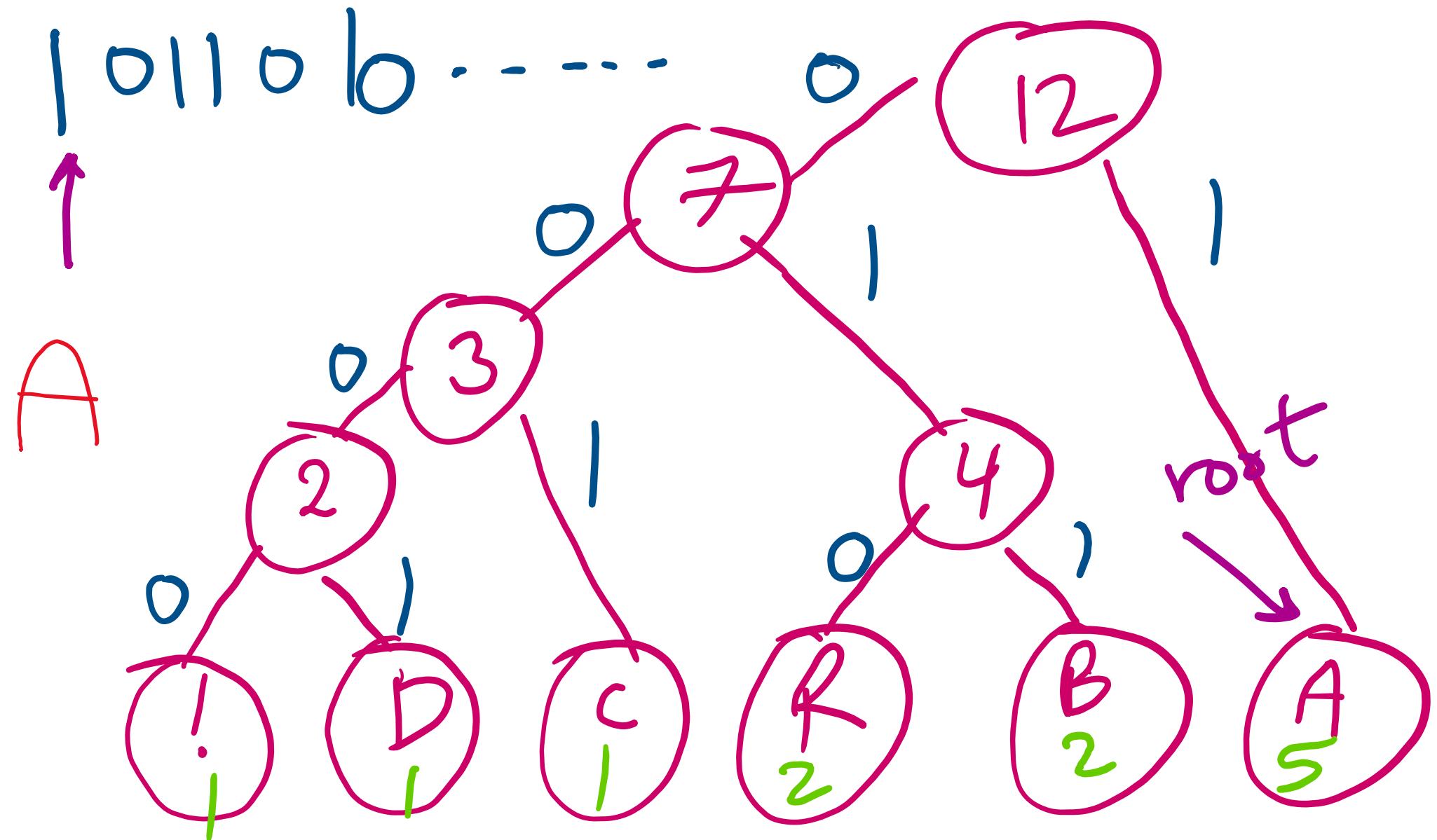
Huffman Compression: Generating the *Codebook*

```
void generateCodeBook(Node root, StringBuilder codeword) {  
    if(root.isLeaf()) {  
        codebook[root.data] = codeword.toString();  
    }  
    if(root.left != null) {  
        //append 0 to codeword, move left, pop last character in codeword  
    }  
    if(root.right != null) {  
        //append 1 to codeword, move right, pop last character in codeword  
    }  
}
```

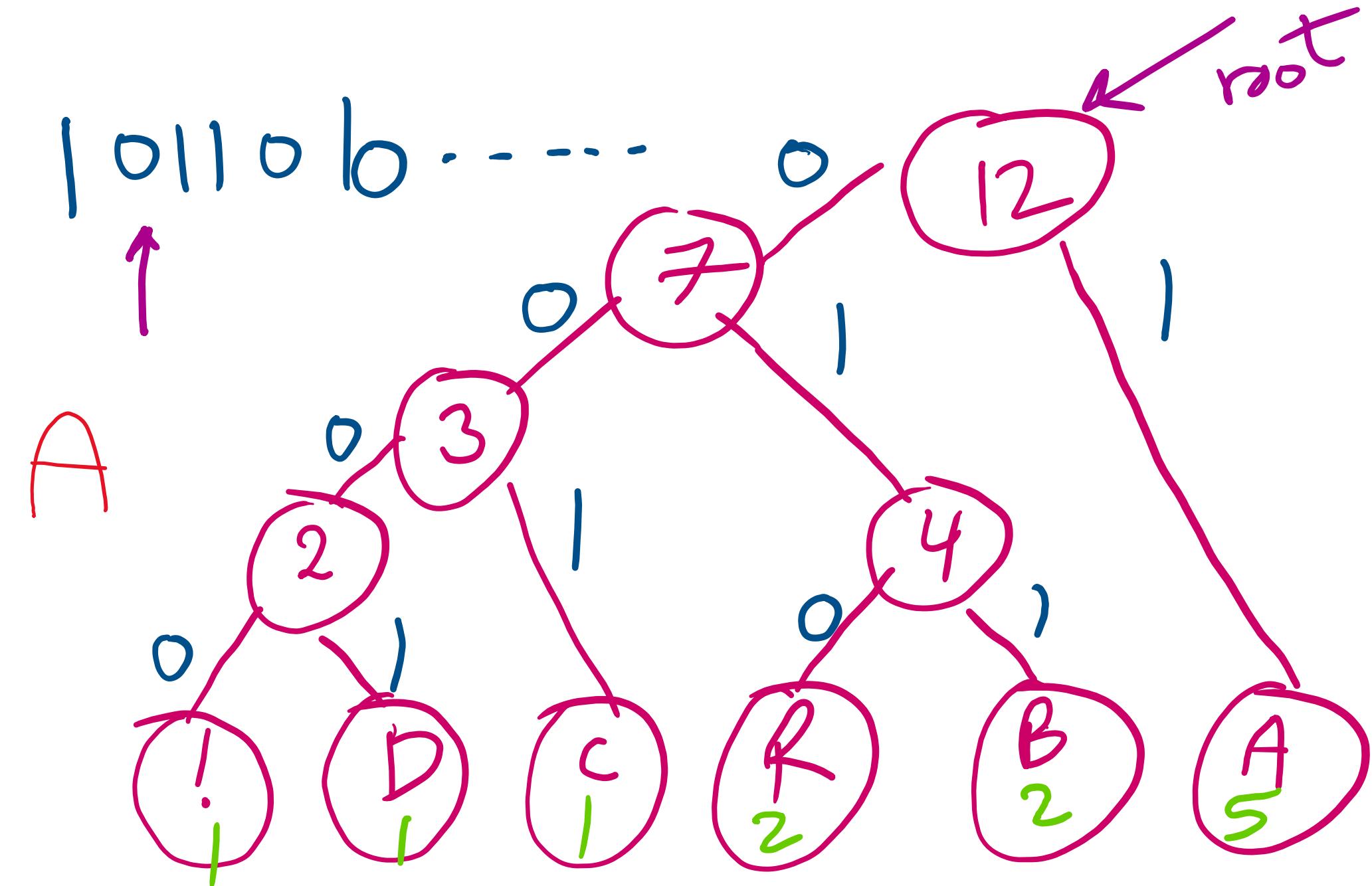
Huffman Compression: Decoding



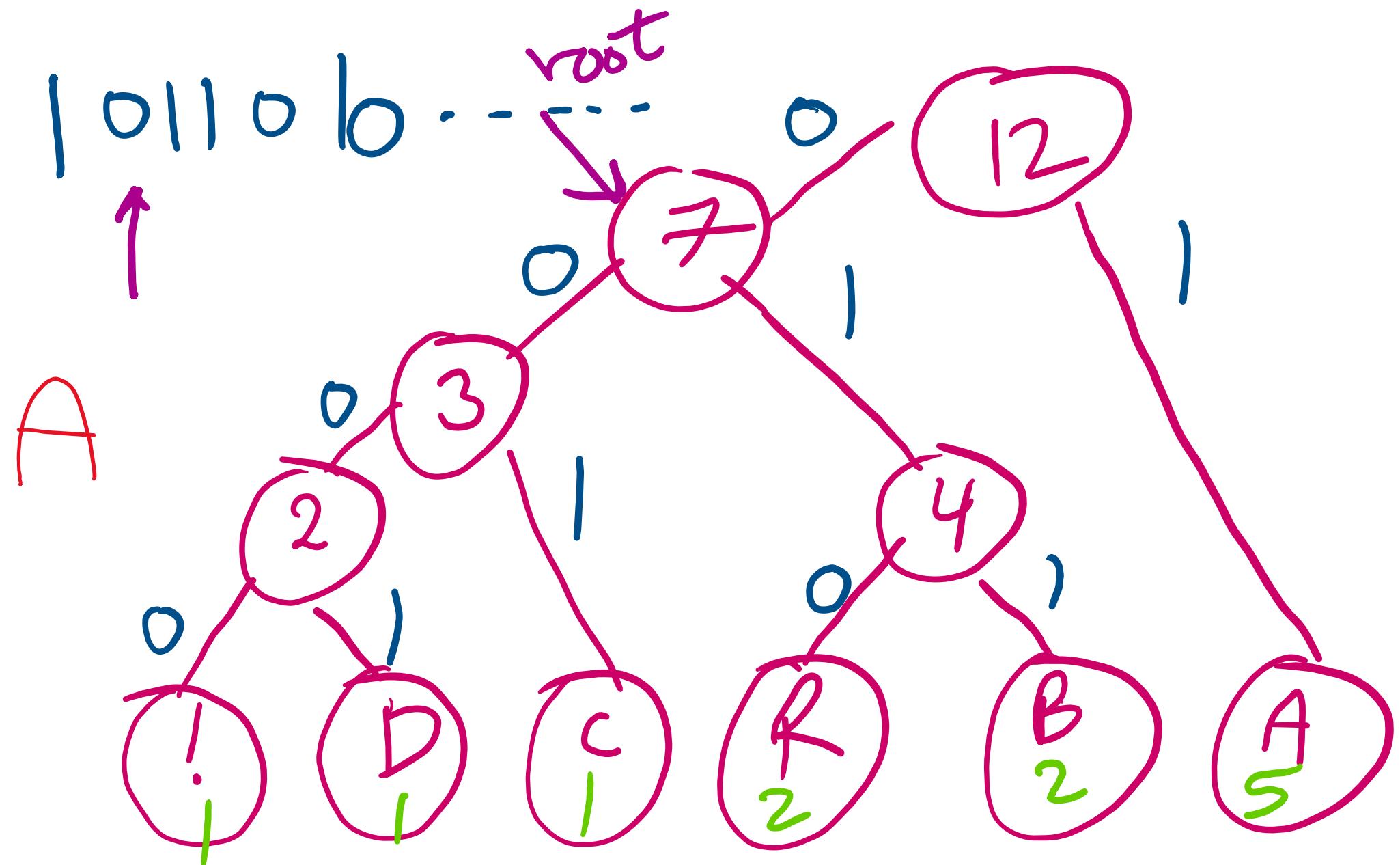
Huffman Compression: Decoding



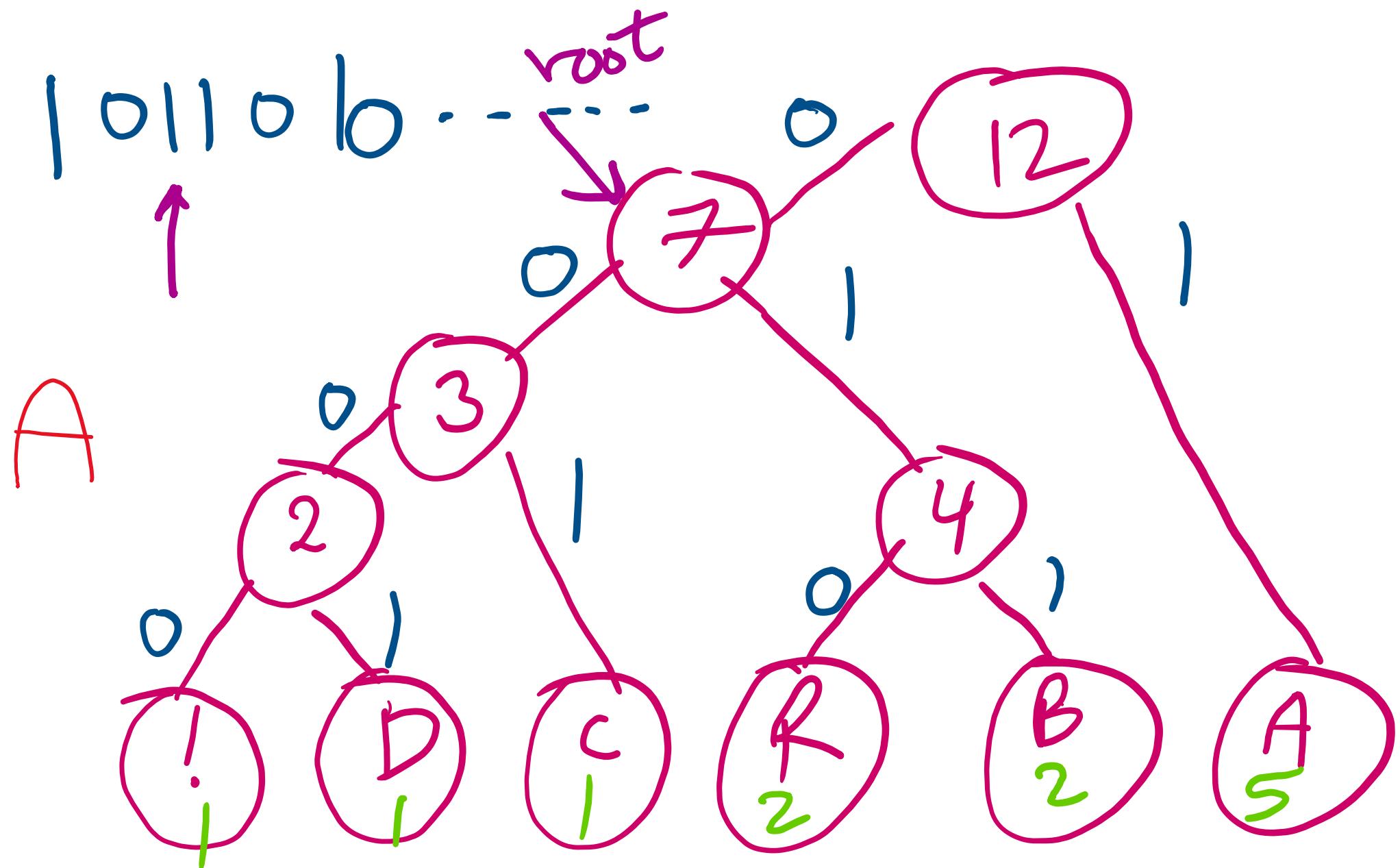
Huffman Compression: Decoding



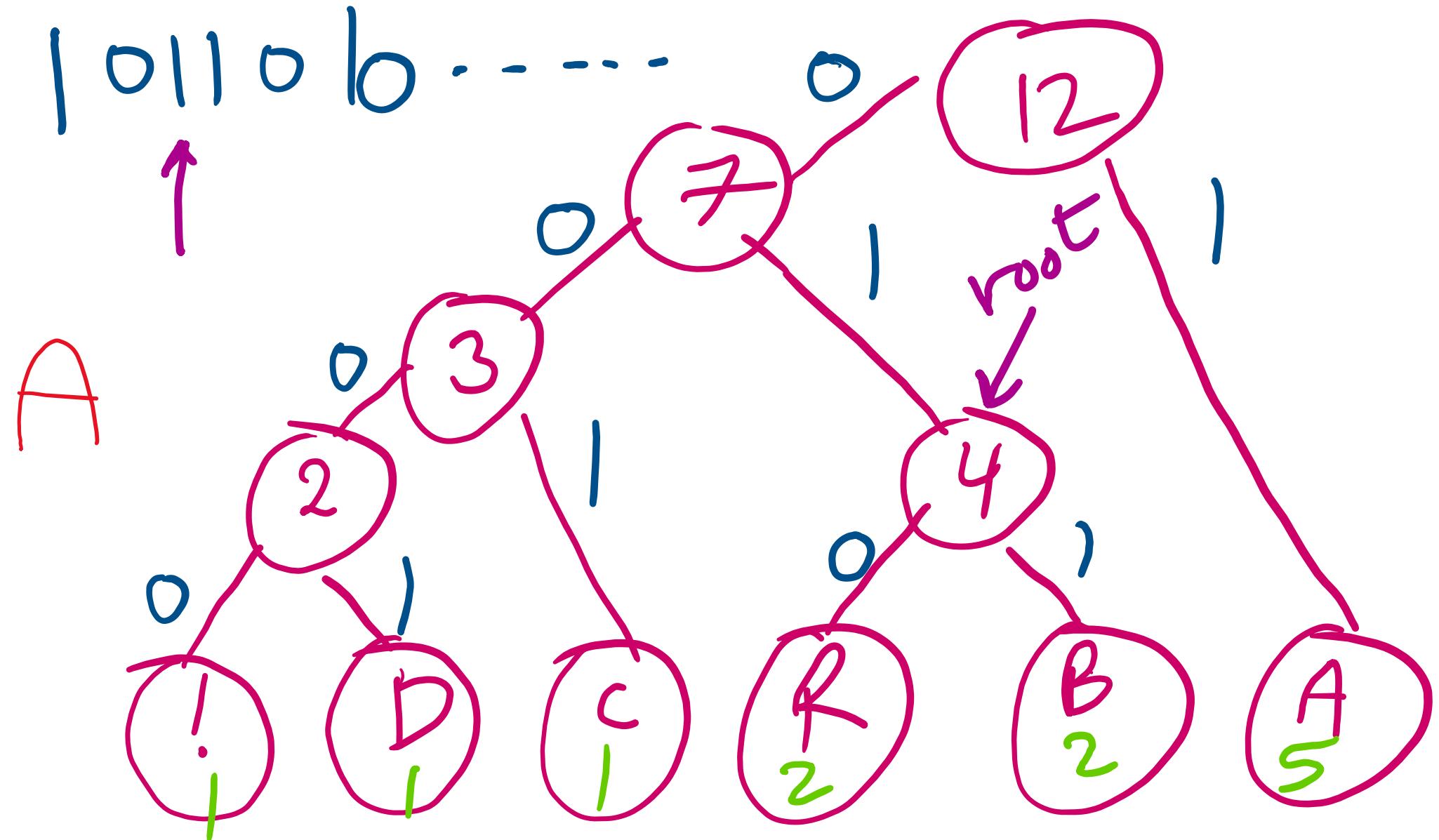
Huffman Compression: Decoding



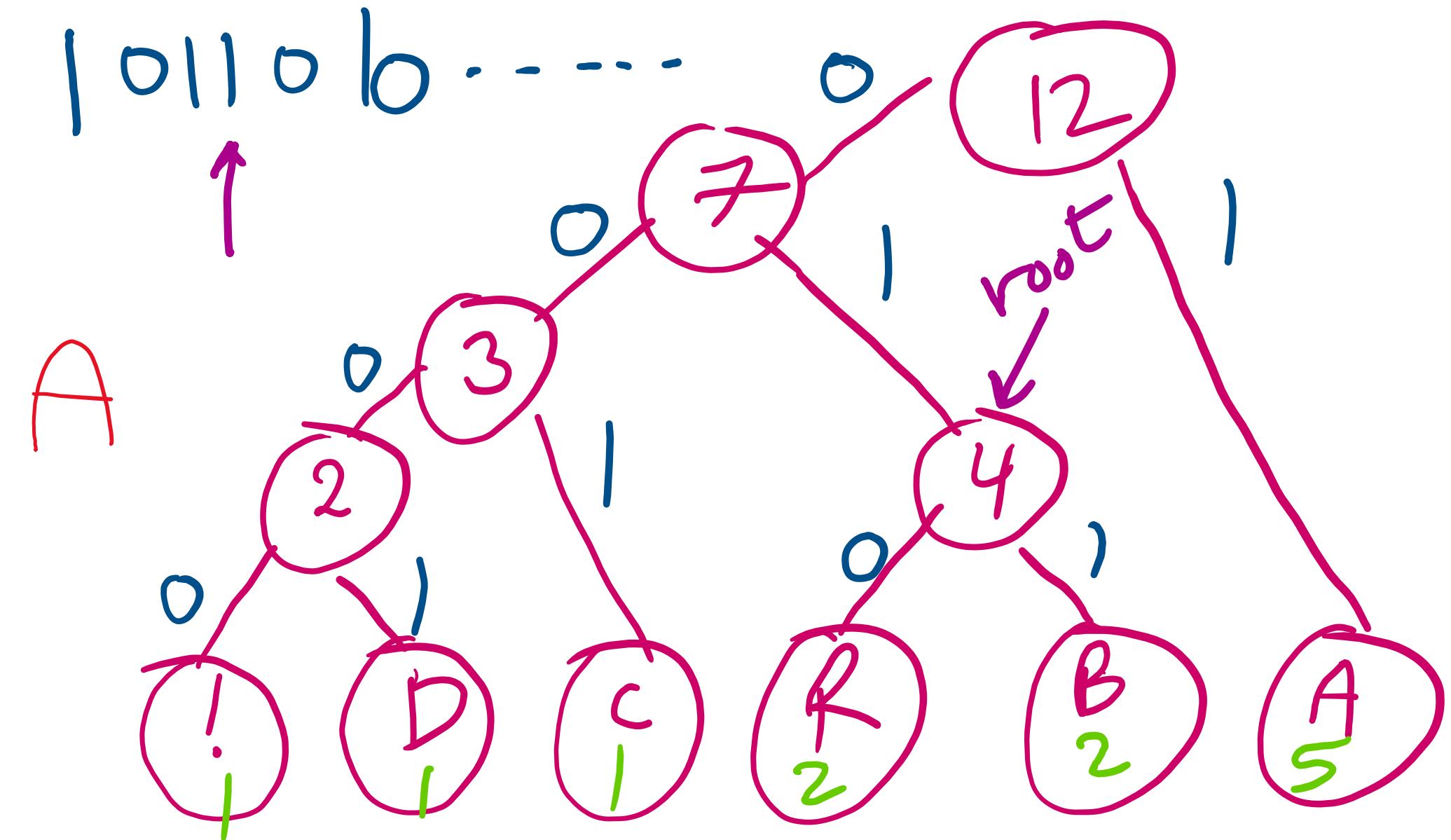
Huffman Compression: Decoding



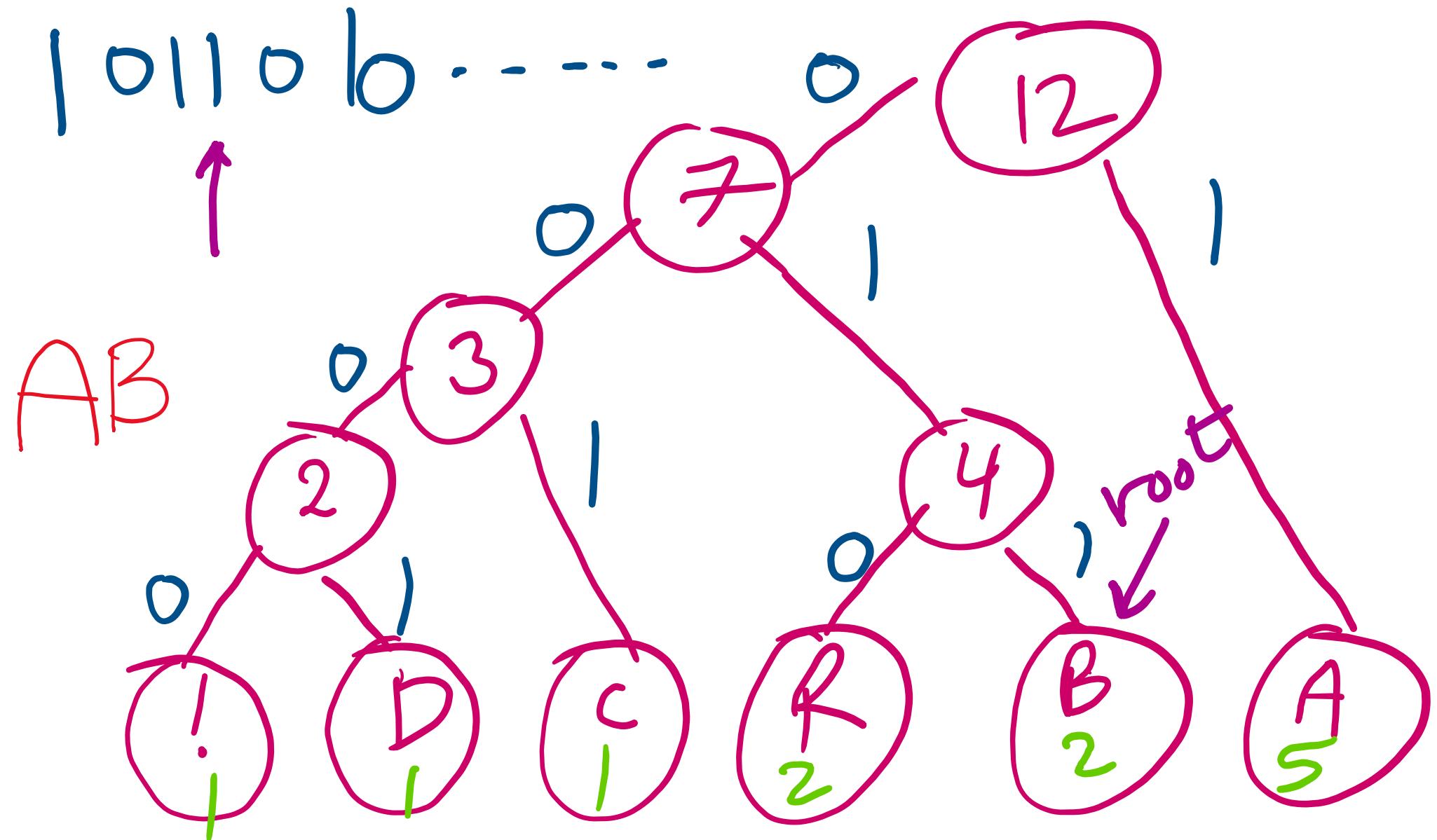
Huffman Compression: Decoding



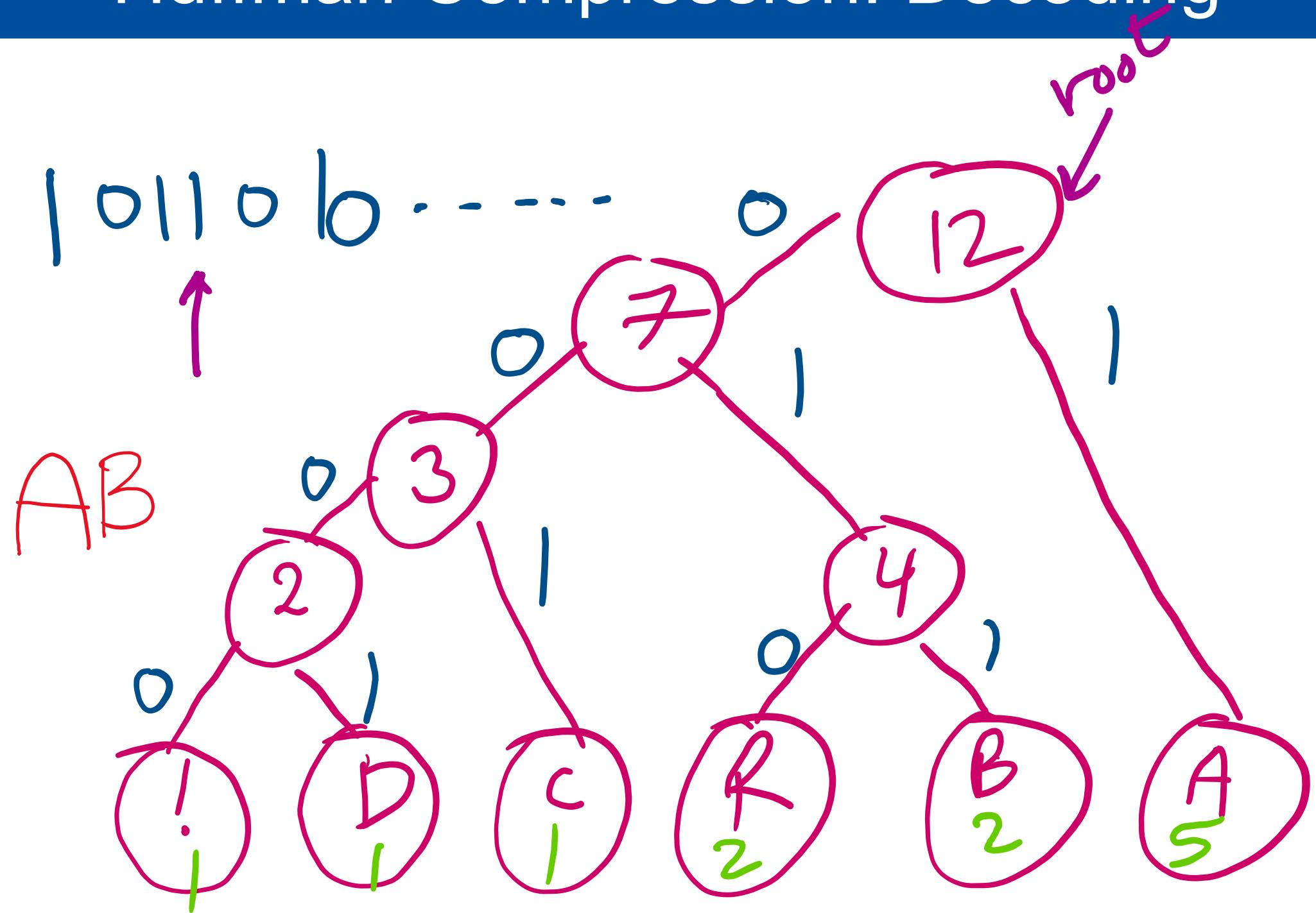
Huffman Compression: Decoding



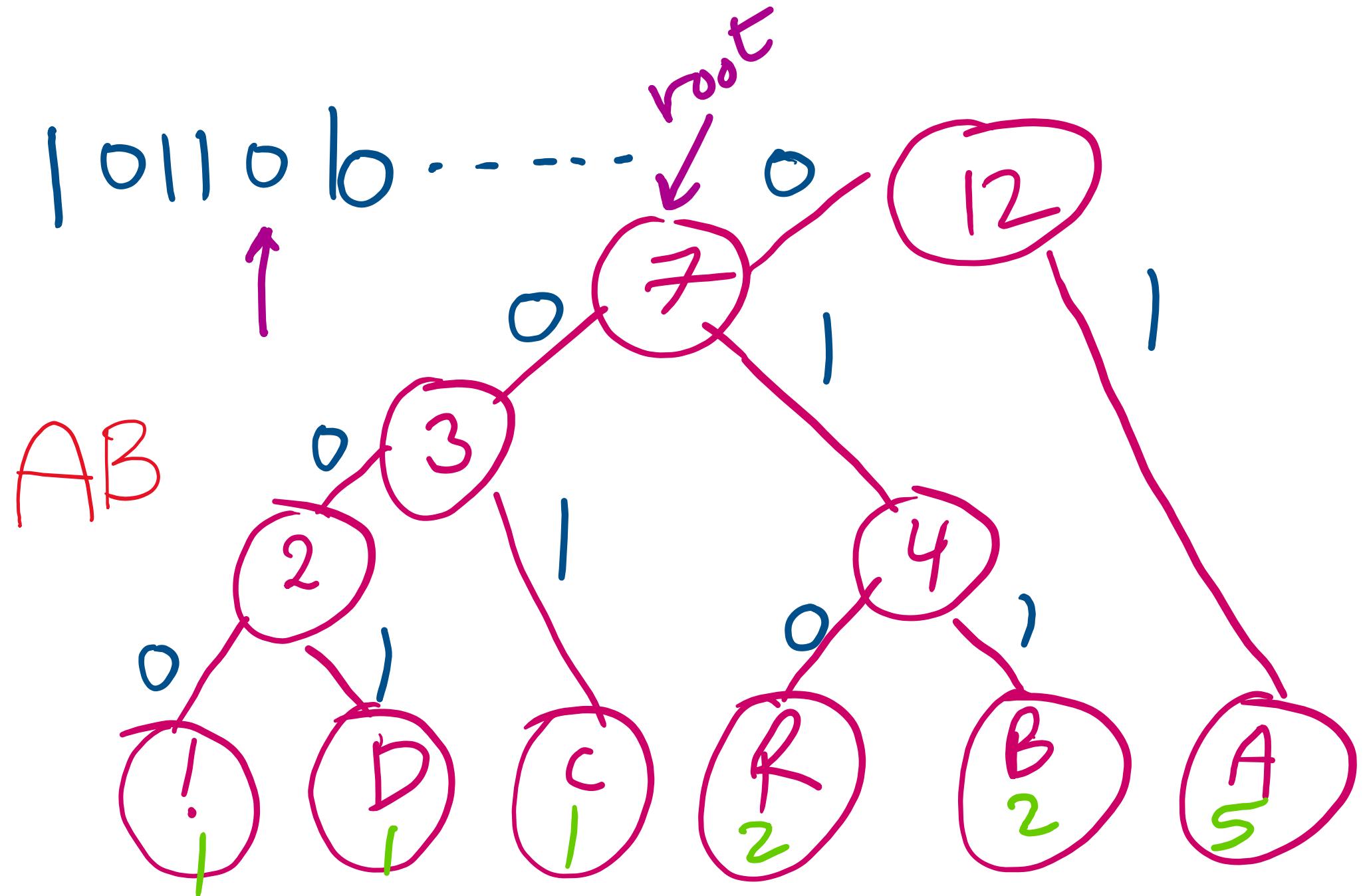
Huffman Compression: Decoding



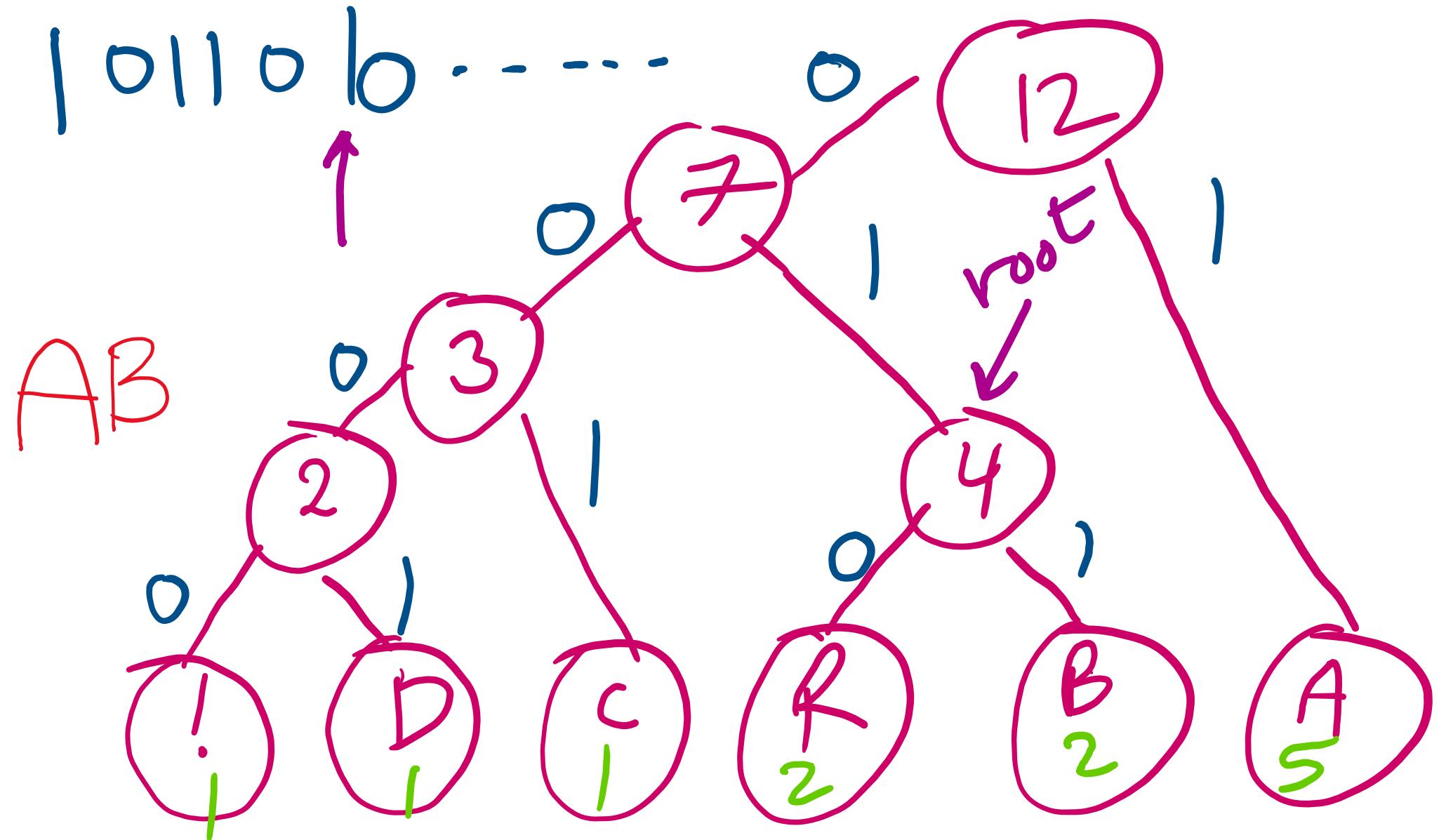
Huffman Compression: Decoding



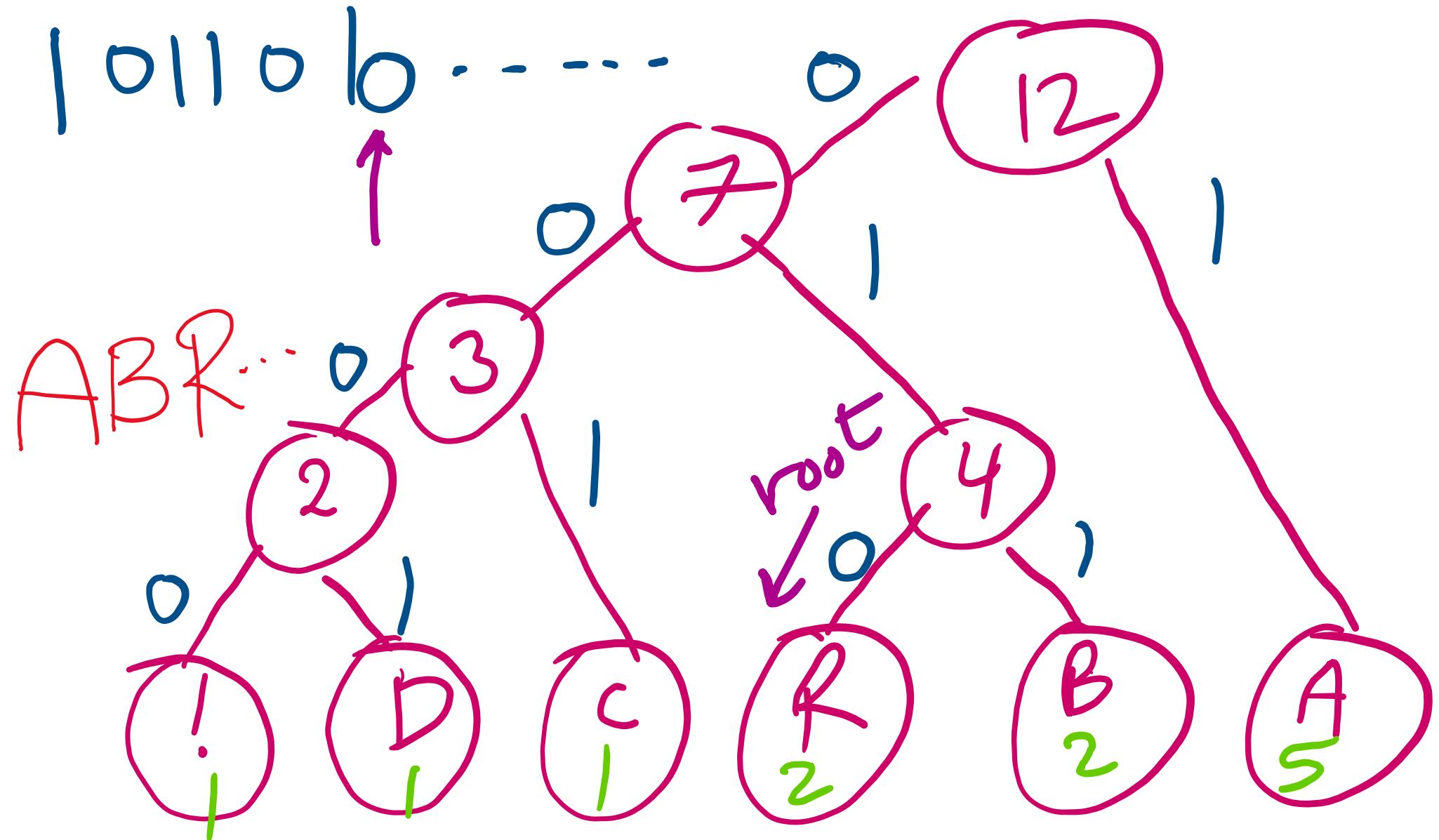
Huffman Compression: Decoding



Huffman Compression: Decoding



Huffman Compression: Decoding



How do we determine character frequencies?

- Option 1: Preprocess the file to be compressed
 - Upside: Ensure that Huffman's algorithm will produce the best output for the given file
 - Downsides:
 - Requires two passes over the input, one to analyze frequencies/build the trie/build the code lookup table, and another to compress the file
 - Trie must be stored with the compressed file, reducing the quality of the compression
 - This especially hurts small files
 - Generally, large files are more amenable to Huffman compression
 - Just because a file is large, however, does not mean that it will compress well!

How do we determine character frequencies?

- Option 2: Use a static trie
 - Analyze multiple sample files, build a single tree that will be used for all compressions/expansions
 - Saves on trie storage overhead...
 - But in general not a very good approach
 - Different character frequency characteristics of different files means that a code set/trie that works well for one file could work very poorly for another
 - Could even cause an increase in file size after “compression”!

How do we determine character frequencies?

- Option 3: Adaptive Huffman coding
 - Single pass over the data to construct the codes and compress a file with no background knowledge of the source distribution
 - Not going to really focus on adaptive Huffman in the class, just pointing out that it exists...

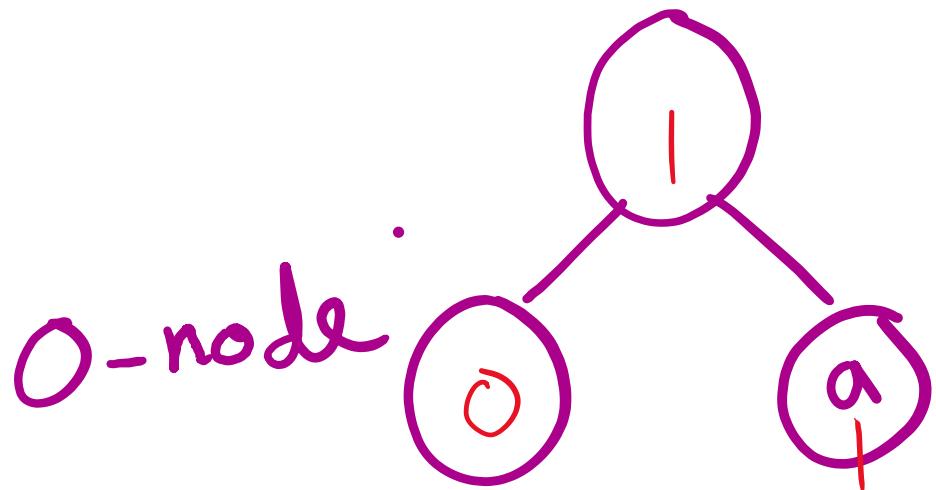
Adaptive Huffman

a a b b c - - - - - - -



Adaptive Huffman

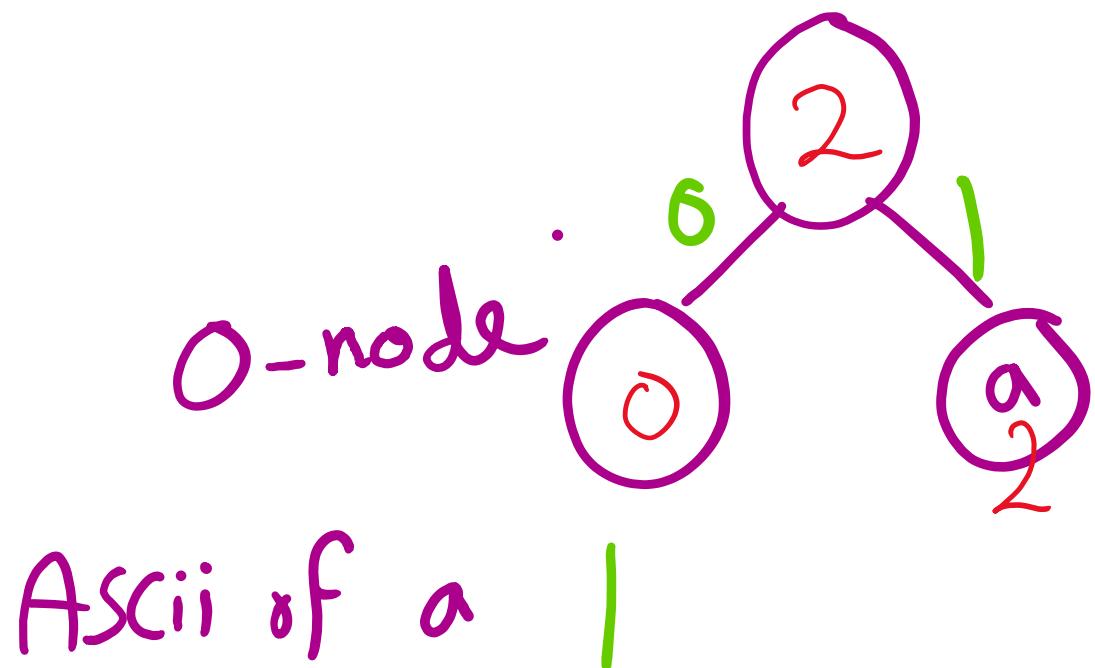
a a b b c - - - - -



Ascii of a

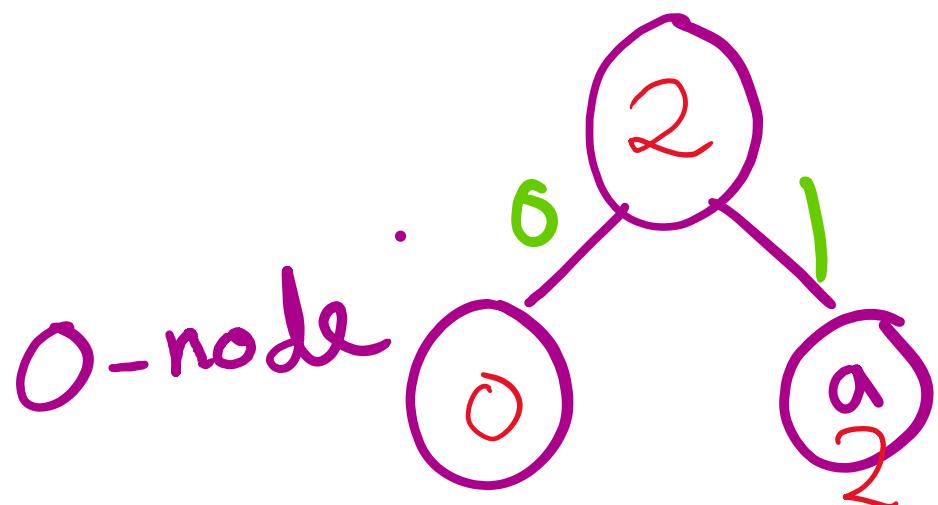
Adaptive Huffman

a a b b c - - - - -



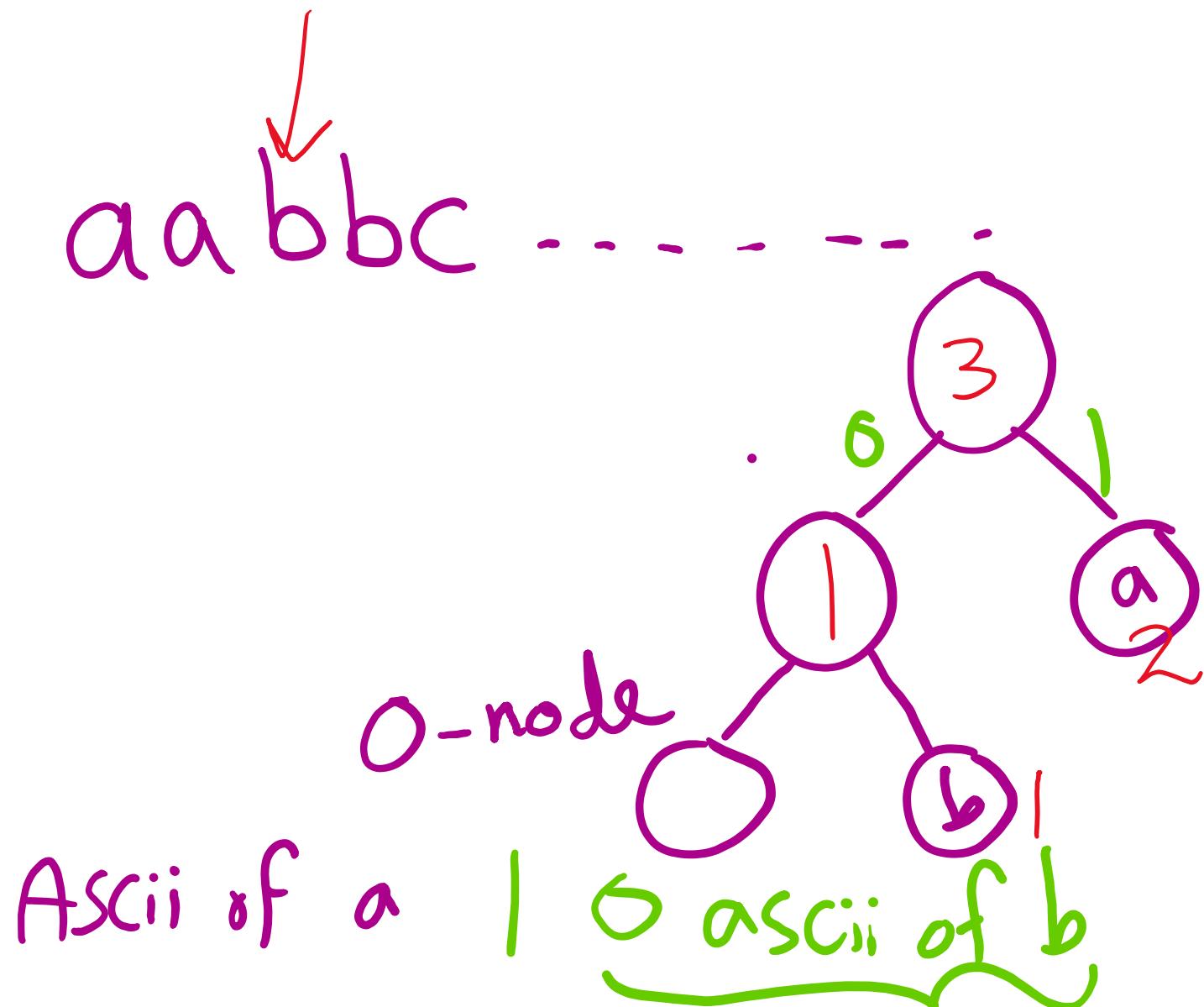
Adaptive Huffman

a a b b c - - - - -

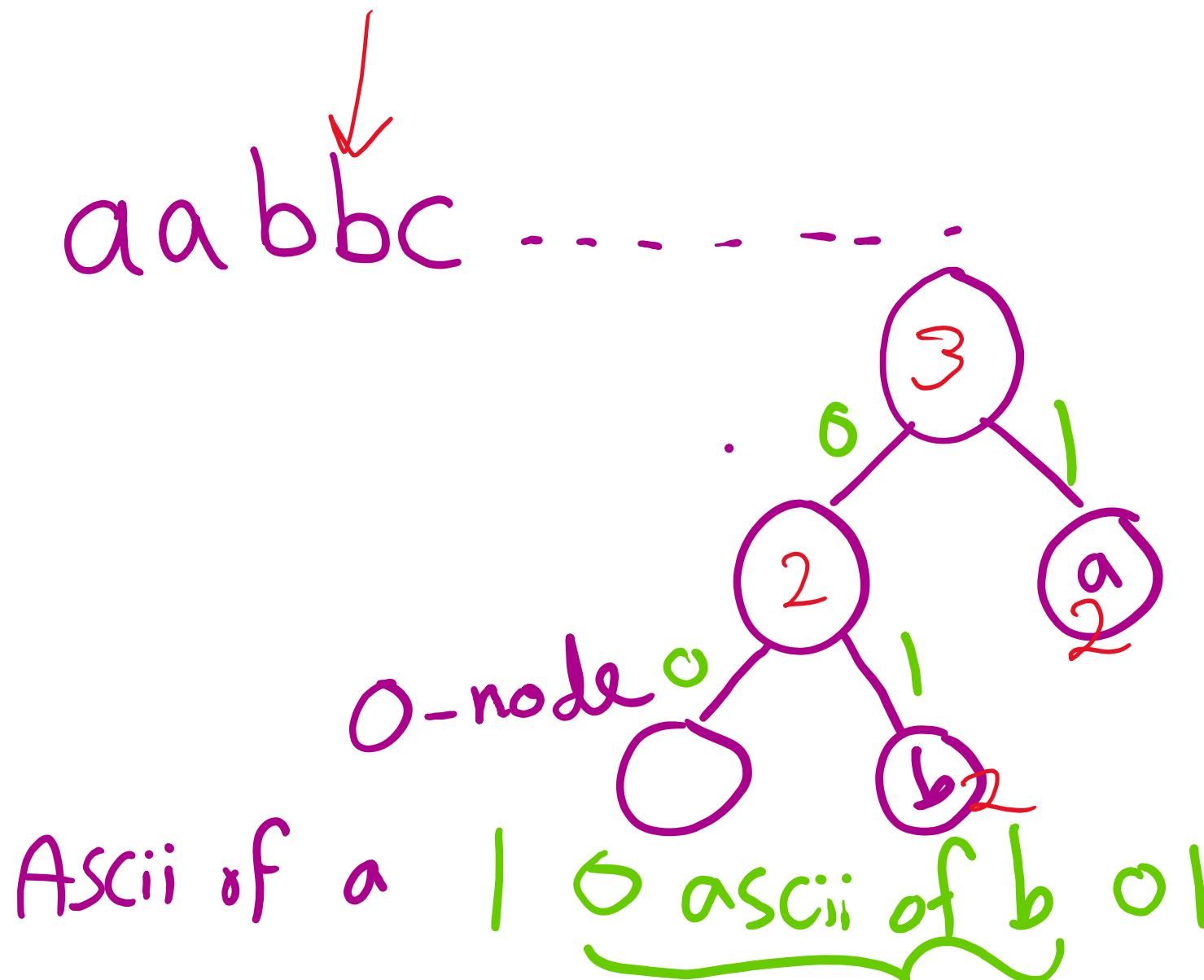


Ascii of a | Ascii of b

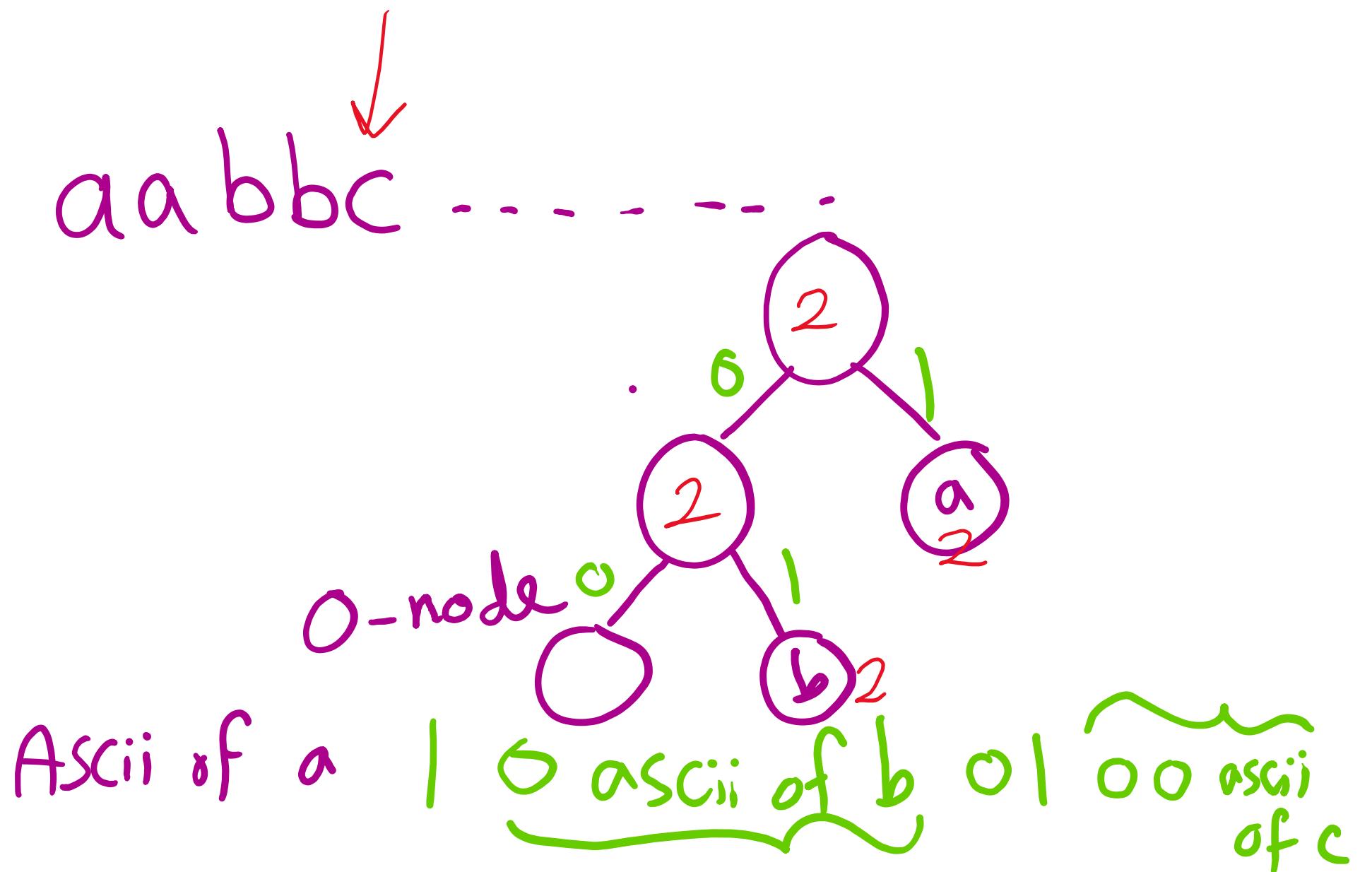
Adaptive Huffman



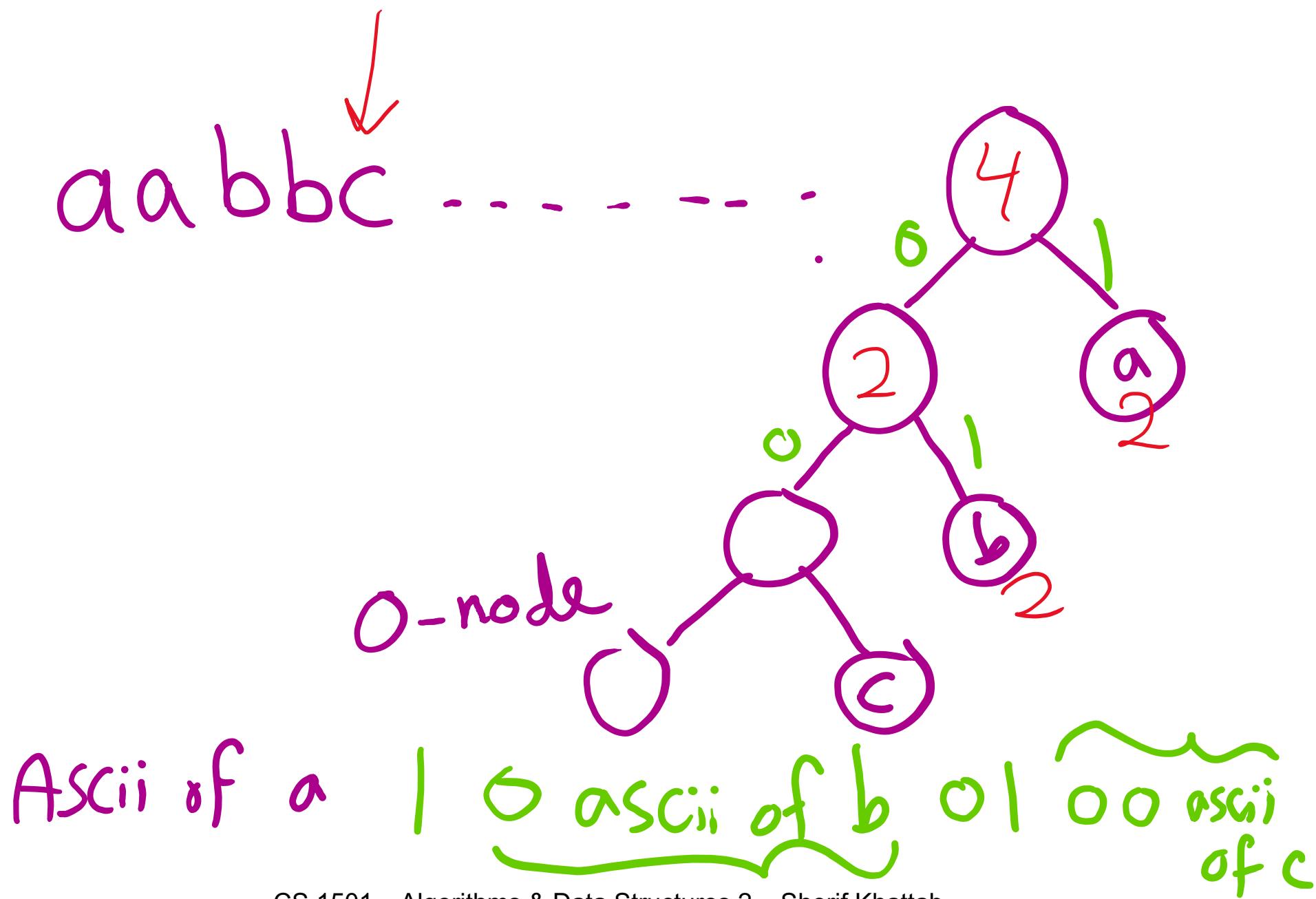
Adaptive Huffman



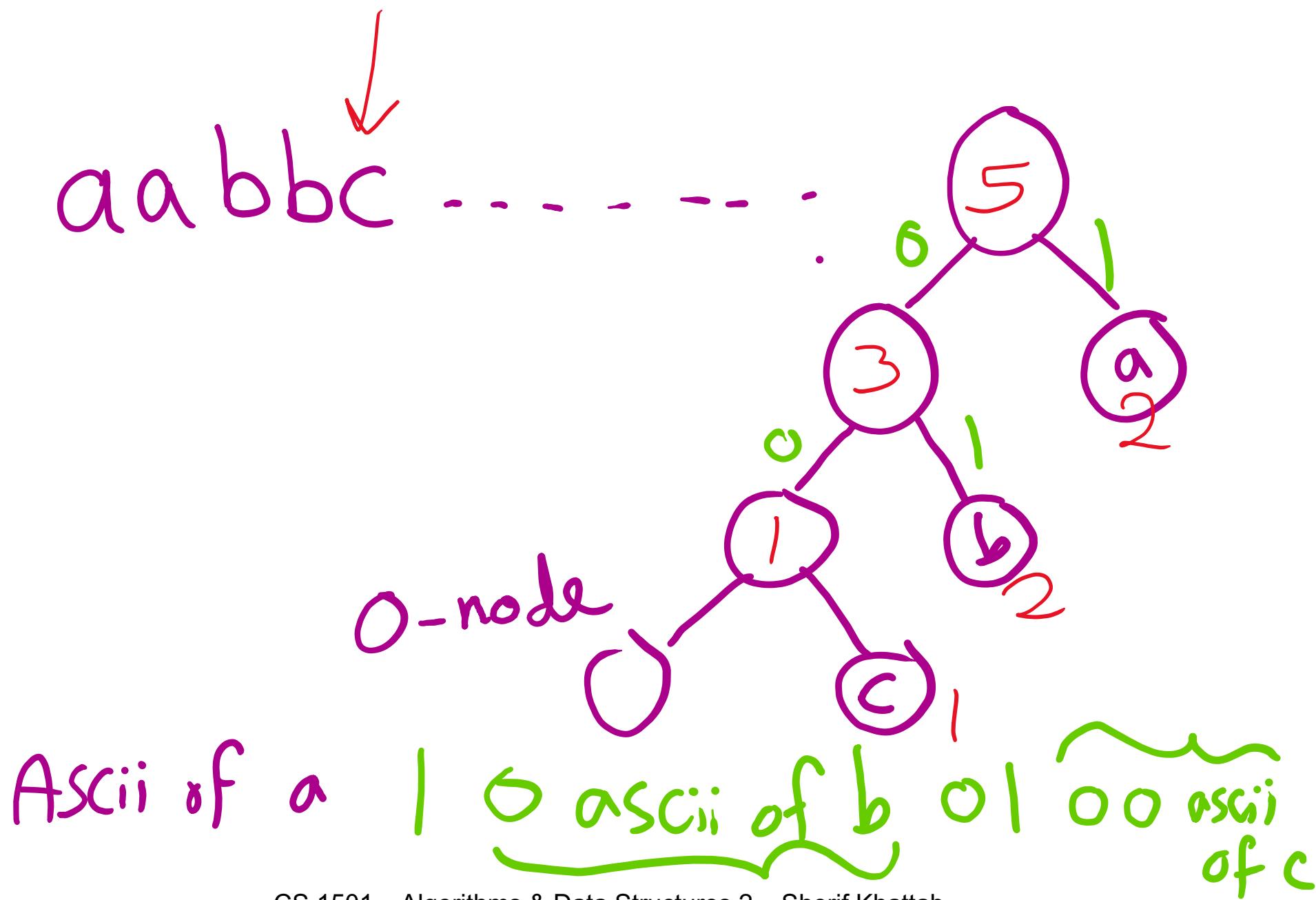
Adaptive Huffman



Adaptive Huffman

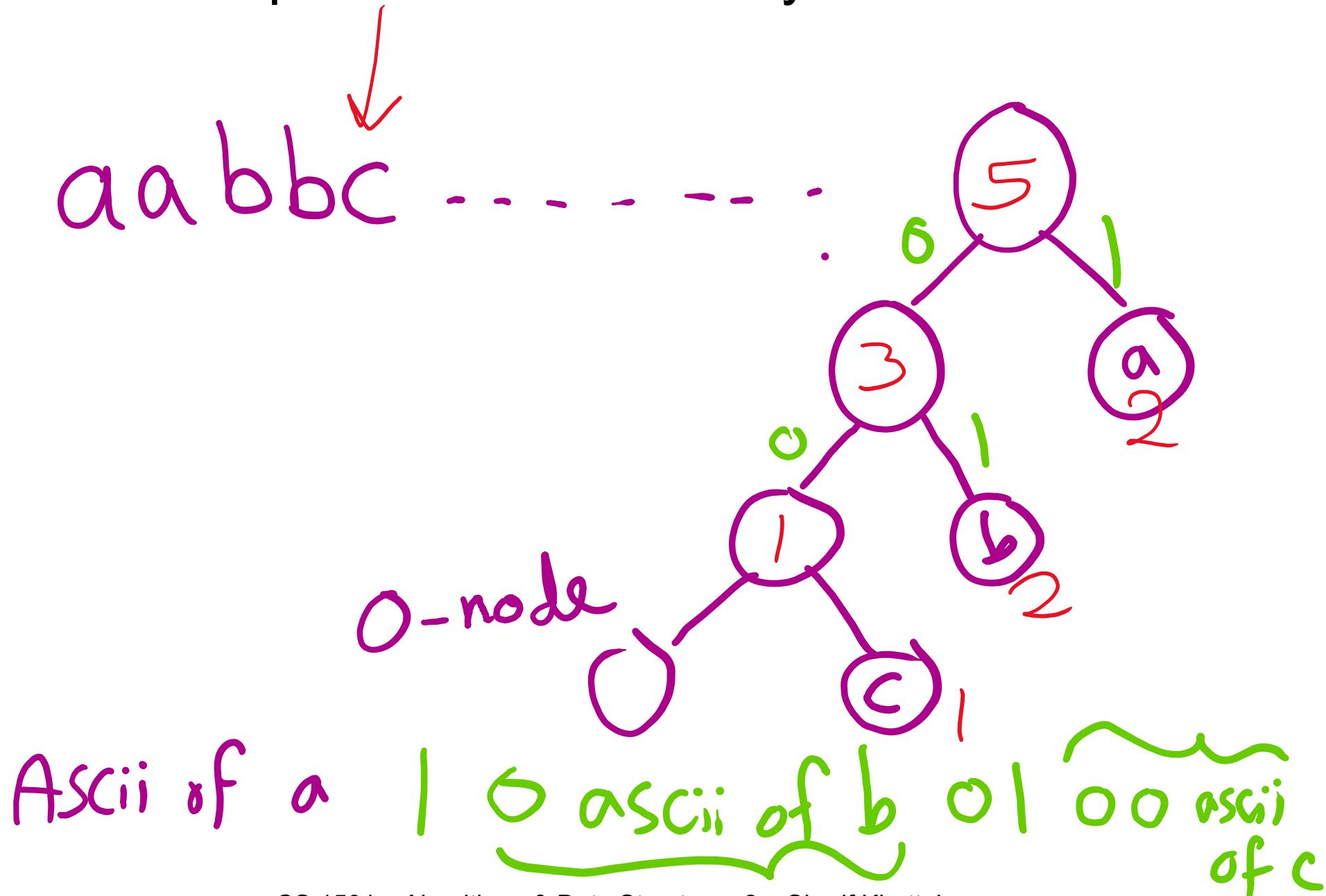


Adaptive Huffman



Adaptive Huffman

- need to swap nodes if necessary



Ok, so how good is Huffman compression

- ASCII requires $8n$ bits to store n characters
- For a file containing c different characters
 - Given Huffman codes $\{h_0, h_1, h_2, \dots, h_{(c-1)}\}$
 - And frequencies $\{f_0, f_1, f_2, \dots, f_{(c-1)}\}$
 - Sum from 0 to $c-1$: $|h_i| * f_i$
- Total storage depends on the differences in frequencies
 - The bigger the differences, the better the potential for compression
- Huffman is optimal for character-by-character prefix-free encodings
 - Proof in Propositions T and U of Section 5.5 of the text
- If all characters in the alphabet have the same usage frequency, we can't beat block code (fixed-size codewords)
 - Huffman reduces to fixed-size codewords
 - On a character by character basis...